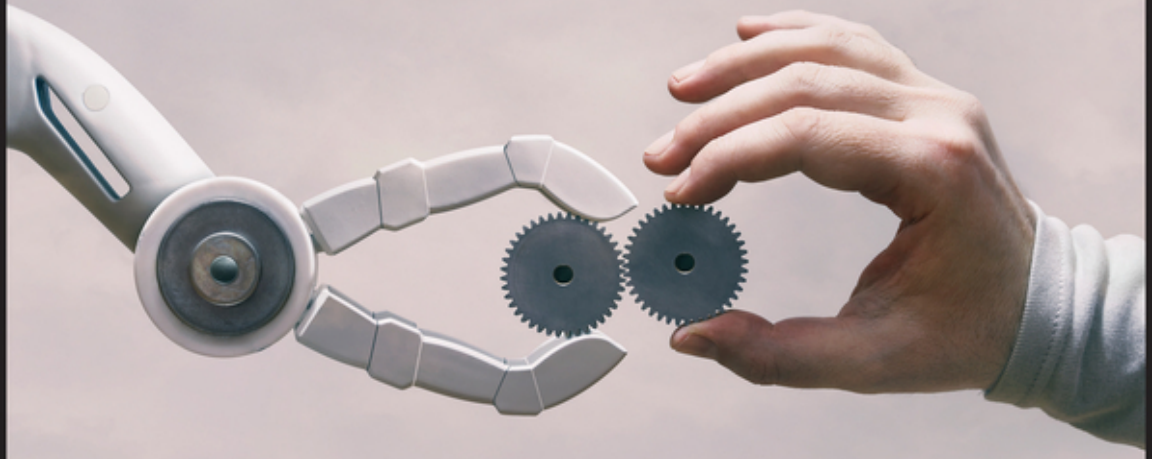
Wiley Series in Systems Engineering and Management
William Rouse, Series Editor

SECOND EDITION

MODEL-BASED SYSTEM ARCHITECTURE

TIM WEILKIENS • JESKO LAMM
STEPHAN ROTH • MARKUS WALKER



WILEY

Model-Based System Architecture

WILEY SERIES IN SYSTEMS ENGINEERING AND MANAGEMENT

William Rouse, Series Editor
Andrew P. Sage, Founding Editor

A complete list of the titles in this series appears at the end of this volume.

Model-Based System Architecture

Second Edition

Tim Weilkiens
Hamburg, Germany

Jesko G. Lamm
Bern, Switzerland

Stephan Roth
Hamburg, Germany

Markus Walker
Ziefen, Switzerland

WILEY

This edition first published 2022
© 2022 John Wiley & Sons, Inc.

Edition History
John Wiley & Sons (1e, 2015)

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by law. Advice on how to obtain permission to reuse material from this title is available at http://www.wiley.com/go/permissions.

The right of Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker to be identified as the authors of this work has been asserted in accordance with law.

Registered Office
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA

Editorial Office
111 River Street, Hoboken, NJ 07030, USA

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Limit of Liability/Disclaimer of Warranty
In view of ongoing research, equipment modifications, changes in governmental regulations, and the constant flow of information relating to the use of medicines, equipment, and devices, the reader is urged to review and evaluate the information provided in the package insert or instructions for each medicine, equipment, or device for, among other things, any changes in the instructions or indication of usage and for added warnings and precautions. While the publisher and authors have used their best efforts in preparing this work, they make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives, written sales materials or promotional statements for this work. The fact that an organization, website, or product is referred to in this work as a citation and/or potential source of further information does not mean that the publisher and authors endorse the information or services the organization, website, or product may provide or recommendations it may make. This work is sold with the understanding that the publisher is not engaged in rendering professional services. The advice and strategies contained herein may not be suitable for your situation. You should consult with a specialist where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Library of Congress Cataloging-in-Publication Data applied for:
ISBN: 9781119746652

Cover Design: Wiley
Cover Image: © Ivan Bajic/Getty Images

Set in 9.5/12.5pt STIXTwoText by Straive, Chennai, India

10 9 8 7 6 5 4 3 2 1

Contents

Foreword *xv*
Preface *xvii*
About the Companion Website *xxi*

1 Introduction *1*

2 An Example: The Scalable Observation and Rescue System *5*

3 Better Products – The Value of Systems Architecting *9*
3.1 The Share of Systems Architecting in Making Better Products *9*
3.2 Benefits that can be Achieved *10*
3.2.1 Benefit for the Customer *10*
3.2.2 Benefit for the Organization *12*
3.3 Benefits that can be Communicated Inside the Organization *14*
3.4 Beneficial Elements of Systems Architecting *15*
3.5 Benefits of Model-Based Systems Architecting *16*

4 Systems, Systems of Systems, and Cyber-Physical Systems *17*
4.1 Definition of "System" *17*
4.1.1 System Elements *19*
4.1.2 System Context *20*
4.1.3 System Characteristics *21*
4.1.4 Purpose *22*
4.1.5 System Evolution *23*
4.2 Definition of "System of Systems" *23*
4.3 Definition of "Cyber-Physical System" *26*
4.4 Composition of a "Cyber-Physical System of Systems" *27*

5 Definition of System Architecture *31*
5.1 What Is Architecture? – Discussion of Some Existing Definitions *31*
5.2 Relations Between Concepts of "System," "Architecture," and "Architecture Description" *33*
5.3 Definition of "Architecture" *35*
5.3.1 Interactions *36*
5.3.2 Principles *37*
5.3.3 Architecture Decisions *37*
5.4 Functional and Physical Architecture *37*
5.5 Taxonomy of Physical Architectures *39*
5.5.1 Logical Architecture *40*
5.5.2 Product Architecture *41*
5.5.3 Base Architecture *41*
5.6 Architecture Landscape for Systems *41*
5.6.1 System Architecture *42*
5.6.2 System Design *43*
5.6.3 Discipline-Specific Architecture and Design *44*

6 Model-Based Systems Architecting *45*

7 Model Governance *51*
7.1 Overview *51*
7.2 Model Governance in Practice *52*

8 Architecture Description *57*
8.1 Architecture Descriptions for Stakeholders *58*
8.2 Definition of "Architecture Description" *60*
8.2.1 Architecture Viewpoints *62*
8.2.2 Architecture Views *65*
8.2.3 Architecture Decisions *67*
8.2.4 Architecture Rationales *69*
8.3 How to Get Architecture Descriptions? *69*
8.3.1 Model-Based Vision *69*
8.3.2 Forms and Templates *71*

9 Architecture Patterns and Principles *75*
9.1 The SYSMOD Zigzag Pattern *76*
9.2 The Base Architecture *82*
9.3 Cohesion and Coupling *85*
9.4 Separation of Definition, Usage, and Run-Time *87*
9.5 Separate Stable from Unstable Parts *89*

9.6 The Ideal System *89*
9.7 View and Model *90*
9.8 Diagram Layout *92*
9.9 System Model Structure *93*
9.10 System Architecture Principles *95*
9.11 Heuristics *95*
9.11.1 Heuristics as a Tool for the System Architect *95*
9.11.2 Simplify, Simplify, Simplify: Strength and Pitfall *97*

10 Model-Based Requirements Engineering and Use Case Analysis *99*
10.1 Requirement and Use Case Definitions *99*
10.2 Model-Based Requirements and Use Case Analysis from the MBSA Viewpoint *102*
10.2.1 Identify and Define Requirements *103*
10.2.2 Specify the System Context *104*
10.2.3 Identify Use Cases *105*
10.2.4 Describe Use Case Flows *109*
10.2.5 Model the Domain Knowledge *110*
10.3 The SAMS Method *112*
10.3.1 SAMS Method Definitions *113*
10.3.2 SAMS Method *114*
10.4 Use Cases 2.0 *117*

11 Perspectives, Viewpoints and Views in System Architecture *119*
11.1 Introduction *119*
11.2 The Functional Perspective *121*
11.2.1 SysML Modeling of Functional Blocks *123*
11.2.2 Architecture Views for the System Architect *124*
11.2.3 Different Architecture Views for the Stakeholders of Different Functions *124*
11.3 The Physical Perspective *125*
11.3.1 Logical Architecture Example *126*
11.3.2 Product Architecture Example *127*
11.4 The Behavioral Perspective *130*
11.5 The Layered Perspective *130*
11.5.1 The Layered Approach *130*
11.5.2 The Layered Perspective in Systems Architecting *132*
11.5.3 Relation to the Domain Knowledge Model *134*
11.5.4 Architecting the Layers *136*

11.5.5 SysML Modeling of Layers *136*
11.6 System Deployment Perspective *142*
11.7 Other Perspectives *144*
11.8 Relation to the System Context *146*
11.8.1 Validity of the System Boundary *146*
11.8.2 Using the System Context as a Part of the Stakeholder-Specific Views *146*
11.8.3 Special System Context View for Verification *147*
11.9 Mapping Different System Elements Across Different Levels *148*
11.9.1 Functional-to-Physical Perspective Mapping *149*
11.9.2 Mapping More Perspectives *153*
11.9.3 Mapping Different Levels *153*
11.10 Traceability *155*
11.11 Perspectives and Architecture Views in Model-based Systems Architecting *155*
11.11.1 Creating Different Architecture Views in a Model-Based Approach *155*
11.11.2 Using SysML for Working with Different Perspectives and Architecture Views *157*
11.11.3 The Importance of Architecture Viewpoints in Model-Based Systems Architecting *159*

12 Typical Architecture Stakeholders *161*
12.1 Overview *161*
12.2 Requirements Engineering *162*
12.3 Verification *163*
12.4 Configuration Management *166*
12.5 Engineering and Information Technology Disciplines *167*
12.6 Project and Product Management *171*
12.7 Risk Managers *174*
12.8 Development Roadmap Planners *174*
12.9 Production and Distribution *177*
12.10 Suppliers *178*
12.11 Marketing and Brand Management *178*
12.12 Management *180*

13 Roles *185*
13.1 Roles *185*
13.2 The System Architect Role *186*
13.2.1 Objective *186*
13.2.2 Responsibilities *186*

13.2.3 Tasks *187*
13.2.4 Competences *188*
13.2.5 Required Skills of a System Architect *188*
13.2.6 Required Skills for Model-Based Systems Architecting *190*
13.3 System Architecture Teams *190*
13.4 System Architecture Stakeholders *192*
13.5 Recruiting System Architecture People *192*
13.6 Talent Development for System Architects *194*

14 Processes *199*
14.1 Systems Architecting Processes *199*
14.1.1 Overview *199*
14.1.2 Example of Generic Process Steps *201*
14.1.3 Example of Concrete Process Steps *202*
14.1.4 Validation, Review, and Approval in a Model-Based Environment *203*
14.2 Design Definition Process *207*
14.3 Change and Configuration Management Processes *207*
14.4 Other Processes Involving the System Architect *207*

15 Tools for the Architect *209*

16 Agile Approaches *213*
16.1 The History of Iterative–Incremental Approaches *214*
16.1.1 Project Mercury (NASA, 1958) *214*
16.1.2 The New New Product Development Game (1986) *215*
16.1.3 Boehm's Spiral Model (1988) *216*
16.1.4 Lean (1945 Onwards) *217*
16.1.5 Dynamic Systems Development Method (DSDM, 1994) *219*
16.1.6 Scrum (1995) *220*
16.2 The Manifesto for Agile Software Development (2001) *221*
16.3 Agile Principles in Systems Engineering *223*
16.3.1 Facilitate Face-to-Face Communication *223*
16.3.2 Create a State of Confidence *224*
16.3.3 Build Transdisciplinary and Self-Organized Teams *225*
16.3.4 Create a Learning Organization *225*
16.3.5 Design, but No Big Design (Up-Front) *226*
16.3.6 Reduce Dependencies *227*
16.3.7 Foster a Positive Error Culture *228*
16.4 Scaling Agile *228*
16.5 System Architects in an Agile Environment *230*

17 The FAS Method *233*
17.1 Motivation *234*
17.2 Functional Architectures for Systems *236*
17.3 How the FAS Method Works *239*
17.4 FAS Heuristics *242*
17.5 FAS with SysML *244*
17.5.1 Identifying Functional Groups *244*
17.5.2 Modeling the Function Structure *246*
17.5.3 Modeling the Functional Architecture *249*
17.6 SysML Modeling Tool Support *250*
17.6.1 Create Initial Functional Groups *251*
17.6.2 Changing and Adding Functional Groups *254*
17.6.3 Creating Functional Blocks and their Interfaces *254*
17.7 Mapping of a Functional Architecture to a Physical Architecture *254*
17.8 Experiences with the FAS Method *256*
17.9 FAS Workshops *258*
17.10 Quality Requirements and the Functional Architecture *259*
17.11 Functional Architectures and the Zigzag Pattern *262*
17.12 CPS-FAS for Cyber-physical Systems *263*

18 Product Lines and Variants *269*
18.1 Definitions Variant Modeling *270*
18.2 Variant Modeling with SysML *271*
18.3 Other Variant Modeling Techniques *276*

19 Architecture Frameworks *279*
19.1 Enterprise Architectures *280*
19.2 Characteristics of System of Systems (SoS) *282*
19.2.1 Emergence *283*
19.3 An Overview of Architecture Frameworks *285*
19.3.1 Zachman Framework™ *285*
19.3.2 The TOGAF® Standard *286*
19.3.3 Federal Enterprise Architecture Framework (FEAF) *288*
19.3.4 Department of Defense Architecture Framework (DoDAF) *289*
19.3.5 Ministry of Defense Architecture Framework (MODAF) *290*
19.3.6 NATO Architecture Framework (NAF) *291*
19.3.7 TRAK *292*
19.3.8 European Space Agency Architectural Framework (ESA-AF) *293*
19.3.9 OMG Unified Architecture Framework® (UAF®) *295*
19.4 System Architecture Framework (SAF) *296*
Together with Michael Leute *296*

19.4.1 SAF and Enterprise Frameworks *296*
19.4.2 SAF Ontology *298*
19.5 What to Do When We Come in Touch With Architecture Frameworks *298*

20 Cross-cutting Concerns *301*
20.1 The Game-Winning Nonfunctional Aspects *301*
20.2 Human System Interaction and Human Factors Engineering *303*
20.3 Risk Management *304*
20.4 Trade Studies *305*
20.5 Budgets *306*

21 Architecture Assessment *307*

22 Making It Work in the Organization *313*
22.1 Overview *313*
22.2 Organizational Structure for Systems Architecting *314*
22.3 Recipes from the Authors' Experience *318*
22.3.1 Be Humble *319*
22.3.2 Appraise the Stakeholders *319*
22.3.3 Care About Organizational Interfaces *319*
22.3.4 Show that it Was Always There *321*
22.3.5 Lead by Good Example *321*
22.3.6 Collect Success Stories and Share them When Appropriate *322*
22.3.7 Acknowledge that Infections Beat Dictated Rollout *323*
22.3.8 Assign the System Architect Role to Yourself *324*
22.3.9 Be a Leader *324*

23 Soft Skills *327*
23.1 It's All About Communication *328*
23.1.1 Losses in Communication *329*
23.1.2 The Anatomy of a Message *330*
23.1.3 Factors Influencing Communication *333*
23.1.3.1 The Language *333*
23.1.3.2 The Media Used *333*
23.1.3.3 Spatial Distance *333*
23.1.3.4 Various Connotations of Words *335*
23.1.4 The Usage of Communication Aids and Tools *335*
23.2 Personality Types *338*
23.2.1 Psychological Types by C. G. Jung *338*
23.2.2 The 4MAT System by Bernice McCarthy *340*

23.3 Team Dynamics *341*
23.4 Diversity and Psychological Safety *342*
23.4.1 Project Aristotle (Google) *342*
23.4.2 Elements of Psychological Safety *343*
23.5 Intercultural Collaboration Skills *344*

24 Outlook: The World After Artificial Intelligence *347*

Appendix A OMG Systems Modeling Language *349*
A.1 Architecture of the Language *350*
A.2 Diagram and Model *352*
A.3 Structure Diagrams *353*
A.3.1 Block Definition Diagram *354*
A.3.2 Internal Block Diagram *357*
A.3.3 Parametric Diagram *361*
A.3.4 Package Diagram *362*
A.4 Behavior Diagrams *363*
A.4.1 Use Case Diagram *364*
A.4.2 Activity Diagram *366*
A.4.3 State Machine Diagram *369*
A.4.4 Sequence Diagram *371*
A.5 Requirements Diagram *372*
A.6 Extension of SysML with Profiles *374*
A.7 Next-Generation Modeling Language SysML v2 *376*

Appendix B The V-Model *381*
B.1 A Brief History of the V-Model or the Systems Engineering Vee *381*
B.2 A Handy Illustration but No Comprehensive Process Description *383*
B.3 Critical Considerations *385*
B.3.1 The V-Model as Process Description *386*
B.3.2 The V-Model Does Not Impose a Waterfall Process *386*
B.3.3 The V-Model Accommodates Iterations *387*
B.3.4 The V-Model Permits Incremental Development *387*
B.3.5 The V-Model and Concurrent Engineering *388*
B.3.6 The V-Model Accommodates Change *388*
B.3.7 The V-Model Permits Early Verification Planning *388*
B.3.8 The V-Model Shows Where to Prevent Dissatisfaction *388*
B.4 Reading Instruction for a Modern Systems Engineering Vee *389*
B.4.1 The Vertical Dimension *389*
B.4.2 The Horizontal Dimension *389*
B.4.3 The Left Side *389*

B.4.4 The Right Side *390*
B.4.5 The Levels *390*
B.4.6 Life Cycle Processes *390*
B.4.7 The Third Dimension *390*

Appendix C Glossary *391*
C.1 Heritage of the Term "Glossary" *391*
C.2 Terms with Specific Meaning *393*

References *399*
Index *417*

Foreword

Contrary to popular myth, models are not new to systems engineering. Models are the way engineers analyze both problems and solutions, so systems models are as old as systems engineering itself. With the traditional focus on written specifications as the "source of truth," models were secondary and descriptive – sometimes reflected as simple sketches, sometimes shown in formal diagrams, partially captured in analysis packages, and often trapped in the mind of the chief engineer. The transformation of systems engineering from document-centric to model-centric practices is not about the introduction of models. It is about making models explicit and moving them to the foreground where they serve as the authoritative tool for design, analysis, communication, and system specification.

Organizations today are investing heavily in representations, standards, methodologies, and technologies to transform the practice of systems engineering through model-driven paradigms. To manage the complexity of today's problems; to keep pace with today's rapidly evolving technologies; to capture the required knowledge regarding the problem, solution, and rationale; to respond effectively to change – all require that systems engineering join the other engineering disciplines in moving beyond document-centric techniques and embracing the power of a model-based foundation. With energy and focus over the last 10 years has come notable progress. The industry has advanced in the area of representations with the development of SysML as a standardized set of diagrams to complement traditional systems representations. Numerous books – including a frequently-cited guide by Tim Weilkiens – explain the details of using this notation to capture and communicate system designs to improve explicitness and alignment within the systems team. Alongside these representations have emerged countless standards and frameworks to help engineering teams develop high fidelity models reflecting key systems dimensions.

However, for all the industry discussion regarding SysML, representations, standards, and tools, there remains a great deal of confusion. Understanding

SysML notation and drawing SysML diagrams do not equate to doing model-based systems engineering. Nor is the use of disjoint models and simulation in systems engineering equivalent to integrated model-based systems engineering.

Effectively moving forward with the transition to model-centric techniques requires that we step back to understand the bigger picture. Diagrams and other representations do not live in isolation but are interrelated and overlapping, communicating key aspects of the system model from specific viewpoints. System architecture and detailed analytical models are not disjoint, nor is there a single grand unified model to capture all dimensions of interest for all systems problems. To move forward, we must embrace the holistic systems perspective and apply it to model-based systems engineering, seeking out the interrelationships and developing a robust toolbox of supporting practices.

In this book, Tim Weilkiens, Jesko Lamm, Stephan Roth, and Markus Walker broaden our vision and expose us to a rich set of perspectives, processes, and methods so that we can develop an effective unified framework for model-based systems architecture. Building upon the existing industry library of textbooks on SysML, this book looks beyond representation to address models, viewpoints, and views as part of a modern approach addressing requirements, behavior, architecture, and more. It connects to a larger framework of processes, methods, and tools key to enabling model-centric practices. And it looks beyond the technical space to the critical cultural dimensions, because the transformation to model-centric techniques is far less a technical challenge than one of organizational change. Addressing the broader framework, Tim, Jesko, Stephan, and Markus bring model-centric practices together to help practitioners develop cohesive system architectures – our one chance in the life of a program to manage complexity, develop resilience, and design in critical concerns such as system security.

There is no doubt that the future of systems engineering is model-based. Document-centric techniques simply are not enough as we grapple with the challenges of today and tomorrow. Those practitioners and organizations who are early adopters in developing a cohesive model-centric framework of processes, methods, and tools will certainly be at a competitive advantage – whether producing products themselves or delivering systems services for others. If, as a profession, we can transform from document-centric to model-based systems engineering and do so with the vision of enabling model-based engineering, we can help transform the larger product lifecycle delivering radical improvements in quality, cost, and time-to-market for the benefit of all.

June 2015

David Long
President, Vitech Corporation
INCOSE President (2014 and 2015)

Preface

Reacting to market needs on time with systems of high quality and marketable costs is a strong competitive advantage. Once a market need has been identified, multiple disciplines are involved in developing a system toward it. They need to collaborate closely and each according to a precise understanding of the own contribution to the system development. Effective communication and the creation of understanding for the whole system-of-interest are keys for the success. Organizations are facing a more and more dynamic environment and, at the same time, an increasing organizational complexity of distributed teams and stakeholders and an increasing technical complexity of more heterogeneous relationships between system components and their environment. This context requests an explicit and sustainable system architecture.

Each of the engineering disciplines contributing to system development needs specific views for obtaining the needed insight. System models enable the creation of consistent sets of stakeholder-specific views. People using them gain a fast and comprehensible understanding of the system they are developing, which can help them choose appropriate solutions for fulfilling the market needs. All the views look at the same data baseline. There is no effort to consolidate redundant data or to clarify misunderstandings of inconsistent information and the costs of resulted errors.

A system architect needs to shape the system architecture well for realizing a successful system. Multiple tasks have to be carried out, each using an effective approach. This book provides a toolbox for the architects for their daily challenges. The scope of the book is a model-based environment, that is either already established and running or planned. The book explains how to use the SysML modeling language in obtaining model-based architecture descriptions. Nevertheless, the concepts are independent of SysML and could also be performed with other modeling languages.

This book is about people, models, and better products, based on our belief that model-based systems architecting produces better products by creating communication and insight for people involved in system development. The book presents a collection of methods and approaches, which we see as ingredients for getting the system architecture work done successfully. We present model-based systems architecting, which we see as a required backbone for excellent system architecture work together with the stakeholders. We will show that involving the stakeholders means much more than running through a formalized review process.

A fundamental principle in system architecture is simplification. Without simple concepts to be communicated to the stakeholders, the system architect will not be understood and thus will fail. We advise you, dear reader, to adopt the principle of simplification and apply it to the multitude of approaches presented in the book. Feel free only to choose the most suitable approaches for your daily work and disregard the others until you are in a situation where they turn out to be the useful ones. The book is a well-stocked toolbox and not a rigid all-or-nothing process for system architects.

Our experience tells us that each organization will have a different focus area and will need different approaches. This is why we have bundled a variety of approaches we have observed being applied successfully in the industry, in the hope that you will find some pieces of information that are suitable exactly to your current activities. We have selected those approaches, which we find easy to apply in daily work and which are important for common model-based system architectures. We do not claim to provide a complete set. Every system architect loves to go to a hardware store to extend her toolbox. And from time to time she has to discard one of her tools when it is no longer appropriate.

The book addresses system architects and their managers as well as engineers who are involved or interested in systems architecting. It is the first comprehensive book that combines the emergent discipline systems architecting with model-based approaches with SysML and puts together puzzle pieces to a complete picture. Highlighted puzzle pieces are:

- functional architectures and the Functional Architecture for Systems (FAS) method by Lamm and Weilkiens to derive the architecture from common-use case analysis
- the integration of the concept of layered architectures from the software discipline in the context of system architectures
- the modeling of system variants
- the whole picture of different architecture kinds like functional, logical, and product architectures and their relationships
- a brief description of SysML and
- a summary of the history of the V-model and recent thinking about it in the appendix

As a typical reader of this book, you may have no time to read all chapters in sequential order. Therefore, we have made the chapters as independent from each other as we could, in order to enable you to read them individually or out of a dedicated sequence when you like inspiration about a certain topic. You can find an on-demand reference about particular topics and get inspiration for directly using the presented approaches in your daily business. The topics are demonstrated using a fictitious robot-based solution for virtual exhibition or other robot-based telepresence tours as an example system.

We like to write texts using gender-fair language. On the other hand, we avoid cluttering the flow of reading by using always both genders in the same sentence. Therefore, we have only used one gender where it was not appropriate to use gender-neutral language. Feel free to replace "he" by "she" and "she" by "he" or whatever is appropriate.

We like to thank the "FAS" and "MkS" working groups of GfSE, German chapter of INCOSE, as well as the "Viewpoints" working group of the same chapter in collaboration with Swiss Society for Systems Engineering (SSSE), Swiss chapter of INCOSE. The work in these groups has provided us with new ideas that can now be found in this book. We thank NoMagic for their support in working with the Cameo tool family that we used to create the SysML models and diagrams we used in multiple chapters of this book. We also thank Erik Solda for allowing us to use the robot example, Martin Ruch for contributing ideas about the assessment of organizational interfaces, and all the colleagues at work who have influenced our way of thinking, helped us with foreign languages in both reading and writing or recommended literature and web links that are today part of the foundations of this book. We furthermore thank numerous people who provided us with advice after we had shown or explained them little fragments of this book to listen to a second opinion.

We like to thank all the supporters of MBSE who believe that MBSE enables the successful development of complex systems – in particular, David Long, who is a great expert of MBSE from the very beginning and has written the foreword.

Finally, we like to thank Brett Kurzman, editor at Wiley, Alex Castro, Kathleen Pagliaro, Bhargavi Natarajan, Sarah Lemore, and Viniprammia Premkumar for their support with the first and second editions of this book.

September 2021 — *Tim Weilkiens, Jesko G. Lamm, Stephan Roth, Markus Walker*
Contributors: Matthias Dänzer, Oliver C. Eichmann,
Ralf God, Michael Leute, Sylvia Melzer

About the Companion Website

This book is accompanied by a companion website:

www.mbse-architecture.com

The website includes:
- High resolution version of all the figures in the book.

1

Introduction

Model-based system architecture (MBSA) combines the two key technologies model-based and systems architecting. Both are major parts of the future state of systems engineering [123].

Many systems result from a evolutionary development. They are driven by their parts and do not emerge from the architecture. The parts could be anything that, in combination, is assembled to a human-made purposeful system. System architecture is followed by a complete system. Often system architecture is referred to the architecture from the perspective of a software architecture in combination with the hardware or the architecture of software-intensive systems [43]. We understand system architecture more holistic and also consider systems without any software, even though systems without any software, are rarely handled with systems engineering processes and MBSA concepts like described in this book. A system architecture is always present. In today and future systems engineering, it is crucial to apply explicit systems architecting for the success of the system project [123]. Chapter 5 defines the term "system architecture" within its context.

Studies clearly show that systems architecting is critical for the performance and success of the system [68]. This is particularly evident for projects that require significant architectural work or rework. Due to more and more dynamic and complex markets and environments, the system architecture must more and more support the changing requirements and requests for radical changes. Chapter 3 lists the benefits of systems architecting.

A system architecture is about establishing solutions that are in line with the directions that guide the organization and checked for feasibility by the corresponding experts, about designing interfaces that are agreed from both sides, and about ensuring that the people who should know the architecture of a system have a common understanding of it. MBSA uses models for enabling the creation of healthy communication around the architecture of the system and for ensuring that the architecture is validated from different points of view. Models are a key

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

tool to be capable of developing complex systems on time and in a feasible quality. Chapter 6 defines the term "model" and MBSA and discusses related terms.

Models are more than graphics. There are even models without any graphical representations. Just the graphics is not modeling but drawing. To create a model, you need the concrete syntax, the abstract syntax, and the semantics, which you find in a modeling language. We use the international standard Systems Modeling Language (SysML) as a language for the system requirements and architecture models. Appendix A gives an overview about SysML, including an outlook on the next-generation modeling language SysML v2. Although we extensively use SysML in this book, our methods and concepts are independent of SysML and could also be implemented by any other modeling language.

The system architect is the one in charge of shaping the system architecture. This is a big responsibility and a big challenge. Organizations developing systems should carefully select people who are allowed to architect the system – and these people's work results will be tightly monitored by stakeholders everywhere in the organization. Chapter 22 describes how systems architecting could be embedded in an organization, and Chapter 12 discusses the interfaces to the stakeholders of systems architecting. In particular, Chapter 10 introduces the adjacent discipline requirements engineering that closely collaborates with the systems architecting. The SYSMOD zigzag pattern presented in Chapter 9 shows the relationship between requirements and architecture and clearly demonstrates the need for a close collaboration. Artifacts of the model-based requirements and use case analysis are important inputs for the system architects, especially to elaborate a functional architecture using the so-called Functional Architectures for Systems (FAS) method.

Chapter 17 is a comprehensive presentation of the FAS method. Functional architectures are built of functions only and are independent of the physical components that implement the functions. The functional architecture is more stable than a physical architecture that depends on the frequently changing technologies. The architecture principle to separate stable from unstable parts is covered in Chapter 9 about architecture patterns and principles.

Besides the functional architecture, we define and discuss further system architecture kinds. The base architecture that fixes the preset technologies and adjusts the scope for innovation, the logical architecture that specifies the technical concepts and principles, and the product architecture that finally specifies the concrete system. All three architecture kinds are physical architectures. The layered architecture is an additional aspect to these architecture kinds and is presented in Chapter 11.

Another additional aspect is the modeling of variants. Variability is increasingly important. The markets are no longer satisfied by commodity products. The market requests customized products that fit personal demands of the customers.

Additionally, global markets with different local environments and policies require different configurations of a system. Chapter 18 presents a model-based concept to specify different product configurations and gives a brief introduction to model-based product line engineering (MBPLE).

The architecture concepts are presented with a consistent example system. The "Virtual Tour" system (VT system) provides virtual visits by driving with camera-equipped robots through a real exhibition. The system is easy to understand and, at the same time, sufficiently complex to demonstrate the systems architecting concepts. The VT system is also part of a rescue and observation system to illustrate a system of systems and cyber-physical systems. The systems are introduced in Chapter 2.

The system architect who thinks that his job is to make a diagram and save it on a shared network drive will most probably fail. Same for the system architects who think they are the bosses of the development staff and can instruct the other engineers. It is neither an archaeological job nor a chief instructor job. Systems architecting is collaborative work that requires communication and soft skills. A basis for a good communication is a common language and media to transport the information. Chapter 8 covers the artifacts of the architecture documentations. In Chapter 19, we extend our scope to system of systems and architecture frameworks.

Typically, engineers are focused on the technology challenges of their job. As mentioned, communication and more general soft skills are getting more and more important capabilities. The engineering disciplines are growing together. For instance, that could be seen by the modern discipline mechatronic. And the worldwide humankind is growing together due to the internet other communication and transportation technologies. In consequence, an engineer has an increasing number of communication relationships. She is no longer successful when she only manages her technology tasks. It is also important to collaborate well with team members, stakeholders, communities, and so on. Chapter 23 gives an introduction about soft skills for engineers.

2

An Example: The Scalable Observation and Rescue System

We need an example system for the demonstration of various techniques to be presented in this book. The example shall be based on one single system with one single purpose, but extensible to be explored in scenarios involving the interaction of multiple systems for a purpose different from the one of the original single system.

Our single-purpose system is based on the very old idea of telepresence robots (e.g. [249]). The concrete system was inspired by an organization called "The Workers." They created a robot system that is called "After Dark" [36], because it is intended to be operated at night, when it is dark – and when almost any museum in town is closed. The system comprises robots that are driving through a closed museum. They carry a lamp to shed light and a camera to capture pictures. When sending the captured pictures to a remote user, the resulting offering is a virtual museum tour (VMT). The described system was demonstrated on 23 August 2014 [139]: After Dark's robots were driving through the gallery "Tate Britain", and people worldwide could watch the streamed camera images. A similar virtual museum tour offering based on a robot was started in The Mob Museum, Las Vegas, in 2016 [258], and the same technology was at least considered for several art museums [41].

Inspired by these systems, we present the "Virtual Museum Tour system" (VMT). Its central subsystem will be a robot as shown in Figure 2.1, intended for realizing a remote user's telepresence in a museum. The presented robot is based on some really existing prototype that was created many years ago during a leisure activity by two students (Erik Solda and Jesko G. Lamm) at Aachen university, Germany. To get back from this historic robot prototype to the example system considered here, please imagine the shown fictitious museum tour robot to be an industrial product with today's technologies onboard: It will use latest artificial intelligence (AI) to be able to navigate autonomously in a museum. But of course the system also comprises servers to control such robots, cloud services to offer

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

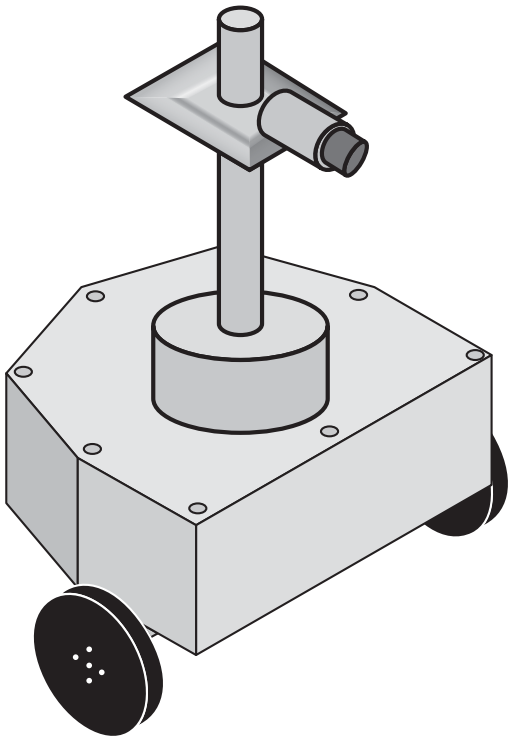


Figure 2.1 The museum tour robot.

onboarding to people worldwide, and apps for mobile devices to schedule virtual museum tours and watch the corresponding video streams.

A storyboard [152] in Figure 2.2 explains the system's main use case: Currently, John is controlling a museum robot to drive it through a museum of arts. He has to write a report about modern art as a homework for school, and he has not had time to go to the museum during its opening hours. John types "Andy Warhol" on his smartphone and the robot starts driving to the pop arts division of the museum. Once there, it stops in the middle of a room. John now selects a painting showing a soup can. The robot moves toward the painting and stops in front of it. The camera on the robot now transmits a picture of the painting to John's smartphone. A little notification box on the smartphone displays the title of the painting. John needs to analyze the artist's way of working in more detail. Via commands entered on his smartphone, he moves the camera down. Then, he zooms in on a particular area of the painting. Now he can see the necessary details via the video stream on his smartphone. This enables John to complete his homework for school.

Unlike the initially mentioned systems at Tate Britain and The Mob Museum, our own example system is purely fictitious and also the extensions to be presented

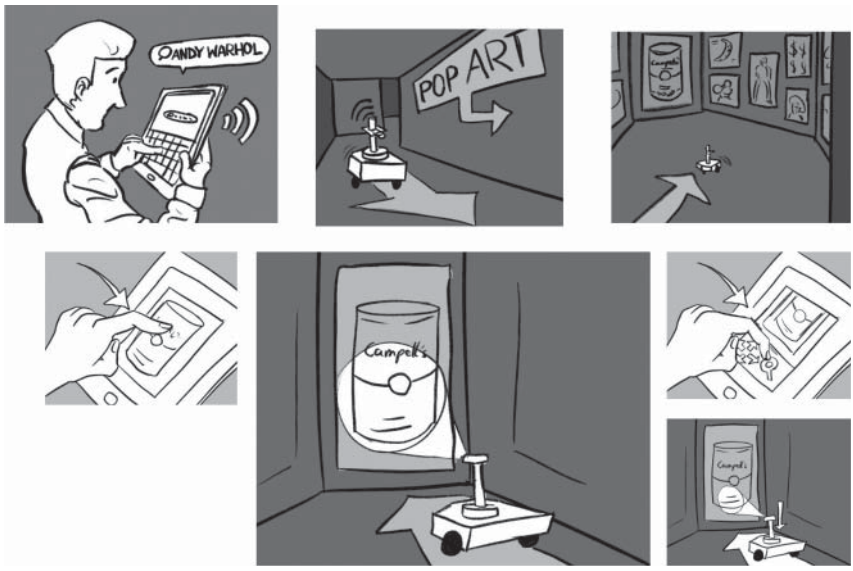


Figure 2.2 Storyboard explaining the Virtual Museum Tour system. ©2020 Jakob K., reproduced with permission.

in the following are freely invented. So, please do not use this book as a construction manual for building such systems (and please mind that the construction of such systems may be constrained by patents filed long before the publication of this book – and even not by the authors of this book). Nevertheless, such systems will be needed to exemplify definitions and modeling.

The fictitious example needs to be extended further, because the VMT use cases we mentioned so far are insufficient for the illustration of some new topics that have newly appeared in this edition of the book. Let's therefore imagine that the VMT manufacturer has found a new business case: Offering company tours in addition to museum tours – enabling companies to show customers around also in dangerous areas of their production site, via a "virtual company tour." The VMT robots are thus developed further, to be used in a more general "Virtual Tour system" (VT). The new system is similar to the VMT and can still be used for museum tours, but its robots are constructed slightly different with more capabilities to navigate in daily business of a production site. They are now called "tour robots" instead of "museum tour robots." Let's furthermore imagine that these enhanced navigation capabilities have attracted the interest of a rescue system provider, a company specialized in leading rescue missions after disasters in production sites, for example those handling dangerous materials. The rescue system provider would then integrate unmanned aerial vehicles (UAVs) and camera

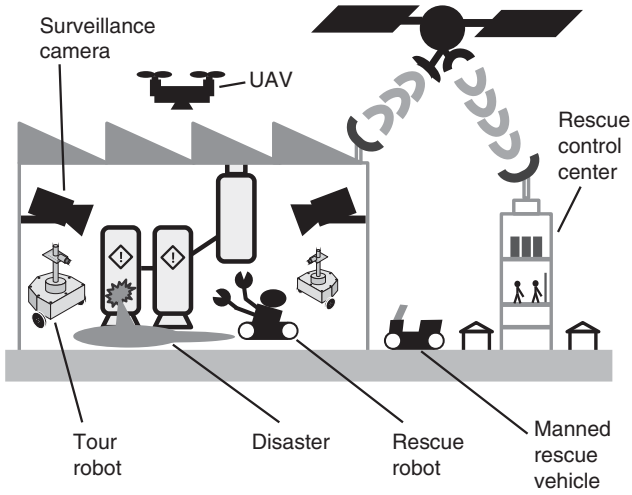


Figure 2.3 The Scalable Observation and Rescue System supporting a rescue mission after a disaster in a chemical production site.

surveillance systems into observation systems and will use these to monitor the situation while working with specialized rescue robots and manned rescue vehicles. A speciality of the observation systems is to be able to integrate the production site's normal video surveillance system into the observation mission. In our invented story, this has led to the idea to also integrate virtual company tour robots into the observation system, because their ability to navigate around makes them more powerful observers than static cameras. The resulting "Scalable Observation and Rescue System" is shown in Figure 2.3.

In the following, we will use the Virtual Tour system as an example, whenever a single system with a single purpose is in scope, and we will discuss the Scalable Observation and Rescue System in cases requiring a more complex example, involving the interaction of multiple systems.

3

Better Products – The Value of Systems Architecting

When driving down to the beach in a nice new car, we may enjoy how well this car grabs the road, and if mentally not yet ready for the beach, we might ask ourselves which department in the car company is responsible for the driver's feeling that the car grabs the road. Is it the suspension design unit or the department with all the steering experts? We believe that all these departments alone cannot make us feel that the car grabs the road, because to do so, the car manufacturer has to see the "car as a system, as a collection of things that interact with each other and with the driver and the road" [166]. Systems architecting will ensure that the interactions between components are controlled in a proper way and that components are designed to fit each other.

3.1 The Share of Systems Architecting in Making Better Products

The example of the car that grabs the road was given by J.N. Martin [166]. We extend this example by speculating what would happen if different developers of car components were asked whose merit it is that the car grabs the road. As a reply, maybe the car manufacturer's suspension department and steering experts as well as the tire companies would claim that they are the ones, by making the best possible suspension, steering, tires, etc., but in contrast to this, consider the following example by Russell L. Ackoff: "Suppose we bring one of each of these [many existing types of automobiles] into a large garage and then employ a number of outstanding automotive engineers to determine which one has the best carburetor. When they have done so, we record the result and ask them to do the same for engines. We continue this process until we have covered all the parts required for an automobile. Then, we ask the engineers to remove and assemble these parts. Would we obtain the best possible automobile? Of course not." [5, p. 18].

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Since systems architecting is concerned with making components fit together instead of making them "the best" each on its own, we believe that systems architecting is an approach that will help an organization think, develop, produce, and maintain better products.

3.2 Benefits that can be Achieved

When talking of better products, then "better" can have two different meanings:

- More satisfying or even more enjoyable for customers (as shown with the "grabs the road" example).
- More profitable for the organization.

Of course, the first aspect may lead to the second one because products that are perceived well by the customer have the potential to become best-sellers and thus to generate profit for the organization.

It cannot be stated in general whether an organization sees it as most beneficial to minimize development cost, production cost, or maximize user satisfaction or certain quality measures. In the end, trade studies will determine the optimum trade-offs between these different criteria and probably many more. Systems architecting is the activity that will produce well-founded trade studies, because it is right in between the requirements analysis work and the solution space that is framed by the development of different system elements. Figure 3.1 illustrates this based on the example of system-level trade studies across the subsystems of the system-of-interest. Systems architecting can enable a top-down realization of business goals, requirements, quality criteria and product strategies into solutions. This is almost independent from whether a completely new system is developed (a very rare case) or whether an existing system needs to evolve further, by architecting increments to an existing solution. The only difference is that constraints from the existing solution will enter trade studies in systems architecting in the latter case, via the expertise from subsystem development.

It is via this top-down approach that usability, maintainability, reliability, etc. can be designed into the system and that concepts like design for user experience [100] or design to market [261] can be realized.

3.2.1 Benefit for the Customer

Different kinds of businesses have different customers: While the consumer goods industry targets millions of individual consumers, subcontractors target industries whose suppliers they are. Despite the variety of customers addressed by different

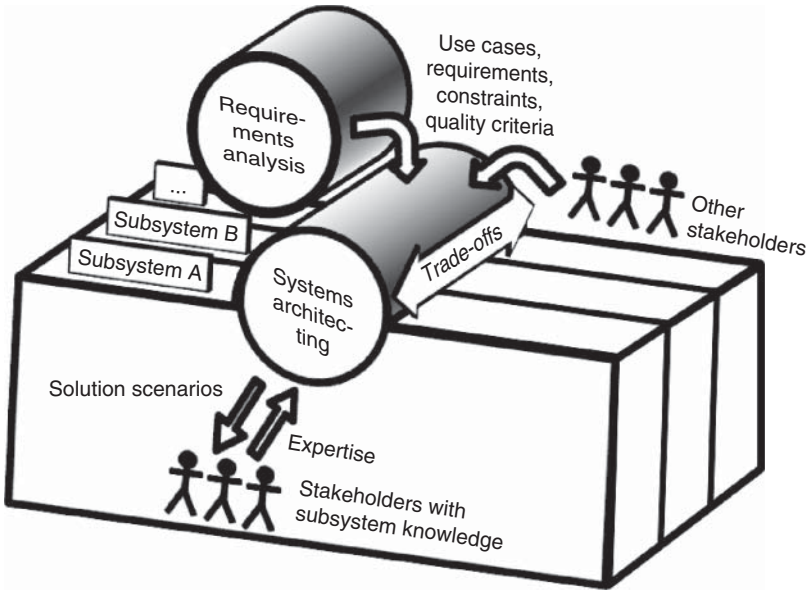


Figure 3.1 Systems architecting plays a major role in breaking down requirements to solutions and in optimizing solutions, for example, by finding good trade-offs.

industries, we believe that any system developer will increase customer satisfaction with investments into systems architecting.

The good news for industrial customers of subcontractors is: We expect change requests and risk management to work very smoothly on a well-architected system with a proper architecture description in place. For example, we have seen cases in which the impact of a change could be easily analyzed, based on the architecture description, and since the system architecture captures dependencies inside the system, we also expect it to help analyzing how uncertainty in one area of the system may lead to risks in other areas. Being able to manage risk based on the knowledge of the system architecture is indeed a potential sales driver, if we believe in one of the conclusions of J.P. Monat's article "Why Customers Buy" [176]: customers' perception of risk seems to be an important factor in the purchasing decision.

The good news for consumers of mass products is: We expect a properly architected system to have a good chance of working as expected in the market place, because systems architecting can allow for thinking the different modes of operation into it instead of just testing them into it. Furthermore, well-defined interface descriptions may offer a basis for planning the systematic review or testing of component interactions in order to find flaws. This is particularly interesting with regard to discovering rare cases for which interfaces are not

prepared. Well-documented architecture also enables continuous reassessment of quality, e.g. by means of reviews. Hence, there is a good basis for preventing flaws introduced by continuous changes during product improvements. This in turn may lower the threshold for continuous changes with the purpose of continuous product improvement.

Last but not least, the properly architected system has a good chance of achieving an attractive cost-benefit ratio, because the organization that developed the system and the production facility producing it could save development and production cost, as it will be explained in Section 3.2.2.

3.2.2 Benefit for the Organization

The standard ISO/IEC/IEEE 42010:2011 [114] declares that each system exhibits an architecture. In other words, every system development produces system architecture. The question is whether the system architecture evolves implicitly or whether it is explicitly defined.

The precondition for the benefits to be discussed is an explicit way of systems architecting, where system development and its strategic planning explicitly involves system architecture processes. In case these are not yet established, there will be an initial investment into architecture work, before the organization can harvest the expected benefits.

Once an organization has established systems architecting as an integral part of system development, it should see the predictability and efficiency of system development increase and it should see cost decrease.

Predictability should be obtained because system architecture supports planning, risk management, and other activities with system-wide scope:

- Planning is supported, because the knowledge about the system's architecture enables a completeness check of the work breakdown for the development work and the identification of dependencies between work packages. It is also supported because the order of integration and the needed verification can be planned and optimized according to knowledge about the system architecture.
- Risk management is supported, e.g. because the system architecture determines the contribution of subsystems to the performance of the system-of-interest and thus needs to be known for quantifying the influence of risks in subsystem development on the overall risk profile of system development. This is applicable for both project risk (the risk of project failure or deviation from its objectives) and product risk (the risk of undesirably or even dangerously wrong product performance).
- Staying in control of cyber-security will only be possible if the system-of-interest as a whole is architected for security, because intruders will find its weakest spot

to attack it. Since the weakest spot may be an interface or a certain interaction pattern of several components, it is important to consider the system as a whole when foreseeing security measures and allocating them to the system's entities.

- Technologies based on artificial intelligence (AI) will require the validation of data used for training the AI, but maybe also the continuous collection of data about the performance of the AI itself. This requires governance across the system about the acquisition and aggregation of data to stay in control of data validity and availability. Control is also necessary about the impact of the AI's output on the performance of the system-of-interest as a whole, best derived from its system architecture. This brings us back to product risk, which can be controlled when such impacts are known.

Good predictability will also be obtained in the more long-term strategic planning, because systems architecting can provide high-level visions and trade-offs based on comprehensive knowledge of the architecture of a system-of-interest and based on the possibility to compare different scenarios based on their impact on the system architecture, even before investing into detailed technology studies.

Efficiency should be gained, because systems architecting can make system development shorter and even lead to reduced production cost. Shorter development time should be achieved because systems architecting can ensure that subsystem development is based on mature interface specifications, reducing the risk of failures during system integration. Production cost can be reduced if production is thought into system development right from the start. Only with a system-level approach, one can effectively avoid errors in the assembly line, like inefficient order of assembly steps. For example, there is a constraint on the order of steps in production if the system is designed to have all the programming interfaces for production inside the housing where they are inaccessible once the system is fully assembled. If production engineers can save cost by using the programming interface after the last assembly step, then a redesign of such a system may be triggered. Systems architecting should ensure that manufacturability is thought into the system in early phases of the life cycle, such that a redesign as described is either not even necessary or happens "on paper" during early concept phases, thus before any money has been invested into engineering the components under redesign.

Cost should decrease, because systems architecting allows to steer system development into a suitable direction right from the start, leading to a focused investment into development efforts. Thinking of Ackhoff's car example from the beginning of this chapter again, one might compute how expensive it can be to build the best possible variant of each part of the car. A very expensive endeavor, particularly because Ackhoff's example tells us that the resulting car will be quite

poor, meaning that even more cost will arise from making the car as a whole better. Systems architecting can help reducing cost by ensuring that subsystems are developed and produced

- according to the correct specification
- in just the quality that is needed and not in worse or (even more expensive) better quality.

We also expect systems architecting to reduce cost because it facilitates communication in and between development teams, spreads knowledge, makes knowledge accessible, and thus ensures that coordination effort stays low. As a result, an investment into systems architecting may lead to reduced cost and thus to return of investment, but also to shortened time-to-market (which can again contribute to return-of-investment, e.g. if the battle against competitors is won in terms of being the first on the market with a certain asset). Of course, this investment into systems architecting is in the first place a driver of cost for organizations not practicing systems architecting yet. New ways of working have to be established via a costly change process. To be able to harvest, it may thus be necessary to accept higher cost in early phases, based on belief in the approach or based on the recognition that other projects have achieved high efficiency after this initial phase.

Finally, it may simply be more satisfactory for the engineers in an organization to work based on a well-defined system architecture and spend their brainpower into making it right in the first place than to run after all interactions inside a system while they are being discovered on the fly. If done right, systems architecting can be fun for all involved stakeholders.

3.3 Benefits that can be Communicated Inside the Organization

In the struggle for budget and resources, different entities in the organization try to create transparency about their contribution to business success. System architects have a particular challenge in doing so, because their contribution is usually indirect. Architecture descriptions cannot be sold to customers, unless the organization sells systems engineering consulting. System architects will thus have to convince the organization of their important contribution to making better products.

Earlier it was mentioned that the suspension department in a car company will claim having a major share in how well the sold cars grab the road. The suspension experts in such a department may be insulted if the organization is told that this was the system architects' merit. Simply saying "System architects are the ones that care about better products" therefore is probably not the right statement to make.

In our experience, the benefits for the organization will in the end convince others of the value of systems architecting. In this case, it is certainly not sufficient to write these on a big poster or present strategy slides that list these benefits. The positive effect of systems architecting has to be experienced in different parts of the organization. The best feedback a system architect can get is a testimonial from a key player in engineering, stating, e.g. how well a certain interface definition activity has made different fields of engineering reach a common understanding and a well-functioning interface agreement.

In our work in systems architecting, we have also experienced that it is the unintended positive side effects of systems architecting that lead to affirmative testimonials from within the organization. How often have we seen the following? We called in several development stakeholders with the intention of making a very small detail discussion. Then, during the course of action, it was revealed that there was a fundamental misunderstanding between two or more parties within the development organization. More than once have some of the authors received feedback like: "this was good, hadn't you called us in then we would have continued our work based on the wrong assumption."

We recommend to record such feedback and to spread affirmative testimonials, because we believe that the strong benefits for the customer and the organization can only be created in an organization where the benefits of systems architecting are well recognized and where different parts of the organization are supporting systems architecting.

3.4 Beneficial Elements of Systems Architecting

The system architect should be aware which deliverables from systems architecting provide benefit to the organization and its customers. There are some deliverables that do not provide the benefit on their own. For example, the system architecture per se does not necessarily provide customer benefits. It is something that is followed by the system and is not explicitly requested by the customer – at least not to a full extend. But the concepts realized in the system architecture can result in customer satisfaction or reducing time-to-market. This can provide benefit for customers or organizations, respectively. The architecture description again does not provide benefits by itself, since it is residing in some repositories and just consuming memory. Only by using the architecture description to increase clarity about the development task at hand can the system architect contribute to avoiding wrong development approaches and flawed products. The system architect thus has to communicate with the system architecture stakeholders to make the organization harvest the benefits of having created a system architecture description.

So in summary, it is what we do around the system architecture and its description that creates the benefit. In turn, this means: before investing any time into a certain systems architecting task, the system architect should assess what to do with the work product from the task. Only if the work product of a systems architecting task can be used for beneficial activities and only if these activities fit into the schedules of the organization should the given task be started.

3.5 Benefits of Model-Based Systems Architecting

The system-of-interest exhibits a system architecture (ISO/IEC/IEEE 42010:2011 [114]). The aim of systems architecting is to shape the architecture that is later exhibited by a system under development. This cannot be accomplished by the architect alone but only by means of a close cooperation between architects and architecture stakeholders, in particular engineering domains and their representatives and developers. If those develop the system according to an agreed architecture description, then the described and exhibited system architecture are aligned.

To ensure that architecture stakeholders have a consistent understanding of the architecture to be exhibited by a system, the architecture description has to be communicated to the architecture stakeholders by means of the appropriate architecture views. Model-based systems architecting allows creating a model from which different views can be generated toward different stakeholders. The model in the background ensures that the different views stay consistent and can be regenerated after major changes.

In a model-based environment, the architect can focus on finding the right views and the suitable visualizations for the work with architecture stakeholders, instead of having to be concerned about keeping different views consistent. The model becomes the single source of truth.

Once a model has been created or updated, new views can be generated on demand. For example, if the system structure is modeled, it is possible to create a list of all system elements from the model at any time and to filter it for certain aspects. One could, for example, create a list of all the user interfaces of the system in order to support the systematic planning of usability testing.

Another aspect of model-based systems architecting is the possibility to validate ideas by means of a model. C. Alexander writes that the way to improve a picture in the mind of a designer is to make an even more abstract picture [13, pp. 77–78]. In terms of model-based system architecture, we suggest that the model takes the role of this more abstract picture as proposed by Alexander. It depends on the task at hand whether the best application of this thought is the creation of executable models that are used for simulations or whether it is the rigorous (human or automated) review of models according to well-defined criteria.

4

Systems, Systems of Systems, and Cyber-Physical Systems*

4.1 Definition of "System"

The term "system" has many, slightly varying definitions, what is not surprising since its usage is very broad. The term origin traces back to ancient Greek times. It originates from the Greek "sustēma" [196]. The statement "the whole is something over and above its parts, and not just the sum of them all" is attributed to the Greek philosopher and polymath Aristotle. In the nineteenth century, the notion of "system" became important with the works in the fields of thermodynamics by Nicolas Carnot and Rudolf Clausius. The General Systems Theory by Ludwig von Bertalanffy relied on a definition of "system." Subsequently, the term was used in psychology, operations research, and systems engineering [58, 265].

The online browsing platform of ISO [119] provides 140+ hits when searching the exact term "system" in the area "Terms & Definitions." We focus hereinafter on three definitions from the systems engineering community.

The standard ISO/IEC/IEEE 15288:2015 [115] defines "system" as a combination of interacting elements which are organized to achieve one or more stated purposes.

The INCOSE Systems Engineering Handbook [265] appends the definition of the standard ISO/IEC/IEEE 15288:2015 with:

> An integrated set of elements, subsystems, or assemblies that accomplish a defined objective.

In the INCOSE brochure "Systems Engineering and System Definitions" [227], the definition of "system" had been broadened. The authors considered a wider field than technical systems:

> A system is an arrangement of parts or elements that together exhibit behavior or meaning that the individual constituents do not.

* Together with Oliver C. Eichmann, Ralf God, Sylvia Melzer, Hamburg University of Technology.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

These definitions have considerable commonalities and differ in some details. The commonalities include that a system comprises multiple interacting building blocks to achieve a goal or something not achievable by the individual building blocks.

Above cited definitions differ in four aspects:

1. The first two definitions consider engineered systems only and therefore consider a purpose or objective that justifies engineering effort to create systems.
2. The building blocks of systems are named differently: "elements," "elements, subsystems, or assemblies," "parts or elements," and "constituents."
3. The building blocks are "organized," "integrated," or "arranged" to form systems.
4. The emergence is addressed differently. The first definition imposes achieving purposes through interaction. The second definition does not state why the integrated building blocks "accomplish a defined objective." And the third relates the emergence of "behavior or meaning" to the togetherness.

Fostering understanding from various viewpoints is a major task of architects. Understanding requires to describe relations between objects, not just individual objects. A best practice for promoting understanding is to capture descriptions in models (for details refer to Chapters 3 and 6). The simplest form of describing relations between definitions is to build n-tuples of terms [6, 276]. Such n-tuples consider terms with obvious relations to each other, such as opposites or terms with similar but still different meaning. Since the relations are obvious, they do not require explicit description. Modeling whole sentences can address less obvious or missing relations or make definitions more accurate [56]. Extending the modeling from sentences to definitions of a whole domain multiplies these benefits. It enables subsequent linking of multiple domains and reduces ambiguity in communication [104]. The definitions provided hereinafter follow such approaches. As outlined in the preface of this book, we use SysML to express and demonstrate the suggested methods and concepts. We use the stereotype «domainBlock» of the SYSMOD profile [271] to describe individual terms. The standard SysML elements association, generalization, and dependency describe the relations among terms.

Figure 4.1 provides a definition for "system" addressing above mentioned differences. This definition only considers engineered systems, the focus of this book. Engineered systems are developed to achieve a purpose within an anticipated operational environment [227]. The building blocks of systems are named "Entity" and take the role as "system element." "System element" is a term also used in the standard ISO/IEC/IEEE 15288:2015 and in the INCOSE Systems Engineering Handbook.

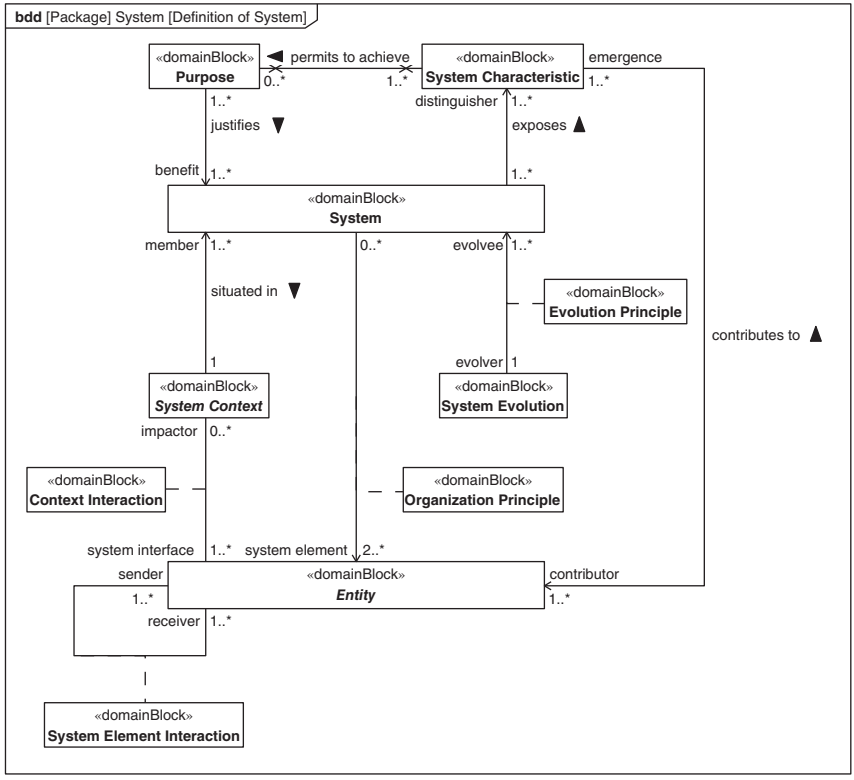


Figure 4.1 Definition of "System."

An engineered system is a composition of multiple entities. The existence of an engineered system is justified by purposes. The exposed system characteristics permit stakeholders to achieve these purposes. Systems are situated in their system context and may be evolved.

4.1.1 System Elements

"System element" designates a role of entities constituting a system. Such entities can be manifold. Figure 4.1 shows system elements as an abstract block named "Entity." System elements interact with other system elements and may interact with the system context. Interactions include exchanging matter, energy, data, or combinations thereof [197]. System elements contribute to system characteristics.

The relation between entities constituting the system and the system itself is depicted with an association block named "Organization Principle." It expresses

that organization principles govern the constitution of systems. The multiplicity on the system side of the association is [0..*]. The lower bound indicates that entities can exist without being integrated into systems. Definition wise the upper bound is not limited. The integration of entities into systems does not impact the properties of these entities. Therefore, the composite association is not navigable from the entity side.

The block of the association block "Organization Principle" is rather an element of the architecture definition, see Chapter 5. This architecture definition element is presented here because SysML imposes to depict association blocks with both, the related association and the related block (here stereotyped with «domainBlock»).

The multiplicity of the entity is [2..*], expressing that at least two entities are needed to form a system. The upper bound is definition wise not limited. To manage the span of control and facilitate understanding, an architect may follow appropriate heuristics and select for instance a limit such as 7 ± 2 [171].

The relations between entities are expressed as association blocks and named "System Element Interaction." The involved roles are named "sender" and "receiver," and both have a multiplicity of [1..*]. At least one entity sends matter, energy, or data and at least one receives these interactions. Definition wise the upper bound is not limited.

System constituting entities may need further decomposition as far as they can be acquired in one or the other way (purchased, manufactured, etc.). Further decomposed entities may be regarded as systems in their own right, also named subsystem or assembly. Entities that can be acquired are considered as leaf entities. This does not preclude that the same entity is considered as a system from another viewpoint. Lawson [156] cites an example where a butcher considers the brain of an animal simply as a kind of meat (a leaf entity), whereas a neurophysiologist considers the same brain as a system. Figure 4.2 depicts the recursive application of the system definition for multiple system hierarchy levels known as composite pattern [84].

4.1.2 System Context

The system context comprises the system under consideration, the system-of-interest [115, 265], and each entity outside of the system boundary that interacts with the system in a non-negligible way. ISO/IEC/IEEE 42010:2011 [114] defines the term "environment" what in this book is defined as "System Context." Many other systems engineering references use "context" or more specific "system context" [104, 189, 226, 265]. The term "environment" may only be connoted to ambient conditions. Figure 4.1 shows that the system context is an abstract block. The focus of Figure 4.1 is on the definition of "system." Therefore, further

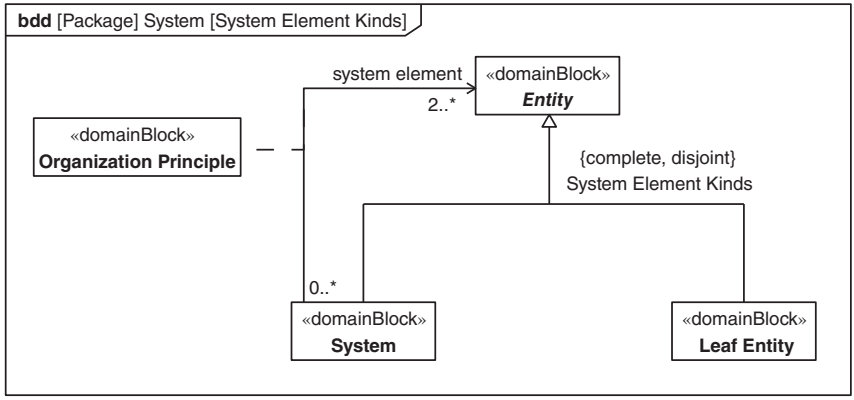


Figure 4.2 Entities playing the role of system elements are regarded either as leaf entities or as systems.

elements of the system context or its variations during the system's life cycle are omitted.

The relation between system and system context is depicted with an association named "situated in." The role of the system is named "member" and has a multiplicity of [1..*]. The upper bound indicates that multiple systems can be members of a system context. Being a member of a system context does not impact the properties of a system; therefore, the composite association is not navigable from the system side. The multiplicity on the system context side of the association is [1]. This multiplicity is to hide the complexity of the system context for this definition of "system."

The relations between system elements and the system context are analogous to those between system elements. The related association block is named "context interaction." The role of the system context is named "impactor" and has a multiplicity of [0..*], expressing that a constituent entity of a system can be impacted or not. Impacted entities play the role "system interface" and have a multiplicity of [1..*]. An engineered system is hardly a closed system. At least one of its elements is exposed to the system context. Other elements may be encapsulated and have no direct impact from the system context.

4.1.3 System Characteristics

System characteristics are distinguishing qualities explaining what systems are or do. Characteristics emerge from the various contributions by the constituent entities of systems.

The relation between system and system characteristic is depicted as association named "exposes," see Figure 4.1. The multiplicity on the system side of the

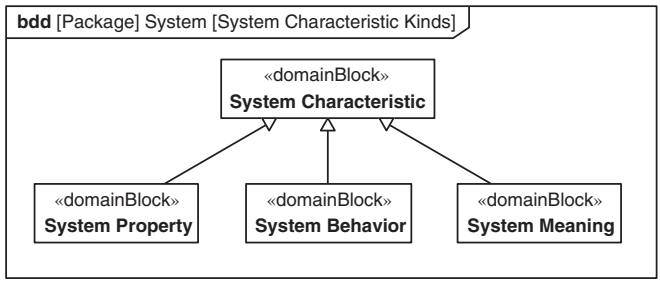


Figure 4.3 System characteristics are the generalization of properties, behaviors, or meanings of the system.

association is [1..*] as individual system characteristics can be observed on multiple systems. The role of system characteristic is named "distinguisher," and its multiplicity is [1..*]. A system exposes one or more system characteristics.

The relation between entity and system characteristic is depicted as association named "contributes to." The constituent entities of systems have the role "contributor" with a multiplicity of [1..*]. For many system characteristics, two or more entities need to contribute. Some characteristics of entities can propagate to system characteristics without contribution by other entities. An example is the length of a housing that encapsulates the rest of a system. The length of the housing also becomes the length of the system. Because system characteristics do not impact the properties of entities, the association is not navigable from the entity side. At least one system characteristic emerges through the contribution of entities. Therefore, the role is named "emergence" and its multiplicity is [1..*].

System characteristics are generalizations of system properties, system behaviors, or system meanings, as depicted in Figure 4.3. System properties are observable or measurable quality attributes. System behaviors are observable responses by the system-of-interest to stimulations. Such stimulations can originate from the system context (e.g. user interaction) or from a system element (e.g. caused by a failure). System meaning is subjective and denotes a significance of the system-of-interest. System meaning relates to a constructionist perspective on systems [156, 227].

4.1.4 Purpose

The purpose points to the use of a system intended by stakeholders attempting to achieve a goal. The system provides benefits in achieving such goals. Stakeholders may intend new purposes during the whole life cycle of systems.

The relation between purpose and system is depicted as association named "justifies," see Figure 4.1. The multiplicity on the purpose side of the association

is [1..*] as multiple purposes can justify a system. The role of system is named "benefit," and its multiplicity is [1..*]. A purpose can justify using multiple systems.

The relation between system characteristic and purpose is depicted as association named "permits to achieve." This association summarizes the relations between purpose, stakeholder, system, and system characteristic as we will see in Chapters 4 and 8 (see Figures 4.1, 8.1, and 8.2). The multiplicity on the system characteristic side of the association is [1..*] as multiple system characteristics permit achieving a purpose. The multiplicity on the purpose side of the association is [0..*]. Some system characteristics do not permit to achieve a purpose because system characteristics can exist to address concerns that are not relevant to justify the use of the system. System characteristics can also unintendedly exist. Neither end of the association imposes properties to the related elements; hence, neither end is navigable.

4.1.5 System Evolution

The system evolution is the gradual development of a system over time. Evolution includes the development, what is considered as the initial evolution [21].

The relation between system and system evolution is expressed as association block and named "Evolution Principle," see Figure 4.1. The role of the system is named "evolvee" with a multiplicity of [1..*]. A system evolution can evolve one or more systems in the same way. The role of the system evolution is named "evolver" with a multiplicity of [1]. Each system has an evolution, at least its development. The evolution principles govern the system evolution. Many engineered systems evolve over time. On conceptual level, the architecture evolves, and on concrete level, existing systems are refurbished.

4.2 Definition of "System of Systems"

Combining multiple systems in a system leads to aggregated systems. If multiple operational and managerial independent systems are integrated into an aggregated system, this special type of aggregated system is denoted as System of Systems (SoS) [162]. The aggregating systems in this specific case are then called Constituent Systems (CS) [53, 130, 265].

Almost regularly, these CS themselves are made up of systems and subsystems. Such type of repeating pattern effect is also known from other domains, e.g. multi-scale modeling of material systems that considers various dimensions of materials and their constituting elements that influence each other and generate the overall material properties. Accordingly, the behavior of SoS is dependent upon the behavior of the contained CS and their subsystems. The repeating

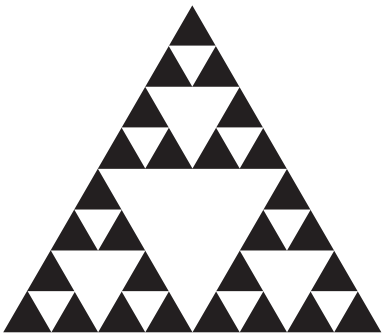


Figure 4.4 Sierpinski triangle representing self-similarity in mathematics. Adapted from [225].

pattern on different hierarchical levels of an SoS complies with the previously described definition in Section 4.1.1 and system characteristics in Section 4.1.3. According to these definitions, systems can comprise entities taking the role of system elements. Since systems are specializations of entities, systems can take the role of system elements as well and be part of a higher-level system. This repeating pattern in system hierarchy leads to the effect, that system structures are repeated on different scales, comparable to the self-similarity effect known from mathematics. An example for demonstrating geometric self-similarity is the Sierpinski triangle in Figure 4.4. Analogously to the geometric repeating structure of triangles in the Sierpinski triangle, there is a repeating structure of systems in SoS.

A historic perspective shows that SoS research exists since the 1950s and can be grouped into three phases. During the first phase lasting for almost 40 years, there is a growing awareness for SoS and the demand for Systems Engineering approaches. Publications in this period describe the need for better understanding of complex systems [35, 228], utilize terms such as "system of cities" that are "systems within systems" [27] and "integrated set" [4], and present first approaches for the development of complex systems [262].

After growing awareness for SoS in the first phase, SoS definitions and characterizations are developed in the second phase. The second phase starts in the 1990s and lasts for approximately one decade. One of the first SoS definitions is provided by Eisner et al. In "RCASSE Rapid Computer-Aided System of Systems (S2) Engineering" Eisner et al. define SoS as "large geographically distributed assemblages developed using centrally directed development efforts in which the component systems and their integration are deliberately, and centrally, planned for a particular purpose." [67]. Eisner et al. describe the systems that form the SoS as component systems. These component systems are denoted as constituent systems (CS) in later definitions. Despite the different nomenclature, Eisner et al. have provided a definition that is still valid today.

In the following years, Maier [162] defines an SoS by emphasizing managerial and operational independence of the component (i.e. constituent) systems:

> A system of systems is an assemblage of components which individually may be regarded as systems, and which possesses two additional properties:
>
> *Operational Independence of the Components*: If the system of systems is disassembled into its component systems, the component systems must be able to usefully operate independently. That is, the components fulfill customer-operator purposes on their own.
> *Managerial Independence of the Components*: The component systems not only can operate independently, they do operate independently. The component systems are separately acquired and integrated but maintain a continuing operational existence independent of the system of systems.

Both definitions from Eisner et al. and Maier state that an SoS consists of components that are themselves systems in turn. The major difference is the independence of the component systems emphasized by Maier. More recent definitions always refer directly or indirectly to Maier's definition, cf. the INCOSE Systems Engineering Handbook version 4 [265] that defines an SoS as "a System-of-Interest whose elements are managerially and/or operationally independent systems."

A third phase of SoS research publications starts in 2000 with the turn of the century and focuses on SoS characteristics and taxonomies. Maier [162] already introduced the SoS types directed, collaborative, and virtual with an increasing degree of managerial and operational independence of the component systems. This taxonomy is expanded by Dahmann and Baldwin in 2008 by so-called acknowledged SoS, that have objectives, a manager, and resources and in that CS keep independent ownership objectives, funding, and development [52]. Dahmann and Baldwin belong to the first researchers that utilize the term constituent instead of component systems. The fundamental four types of SoS, i.e. directed, acknowledged, collaborative, and virtual, are generally accepted today and form the basis of the standard ISO/IEC/IEEE 21841:2019 with the title "Systems and software engineering — Taxonomy of systems of systems" [109]. In this book, we use the definition from the INCOSE Systems Engineering Handbook version 4 based on the definition by Maier [162] that defines an SoS as "an SOI [system-of-interest] whose elements are managerially and/or operationally independent systems. These interoperating and/or integrated collections of constituent systems usually produce results unachievable by the individual systems alone."

The current megatrend of connectivity [170] resulting from digital communication technologies enables the genesis of a new quality of SoS. Previously mostly isolated and centralized systems with few interfaces to other systems are

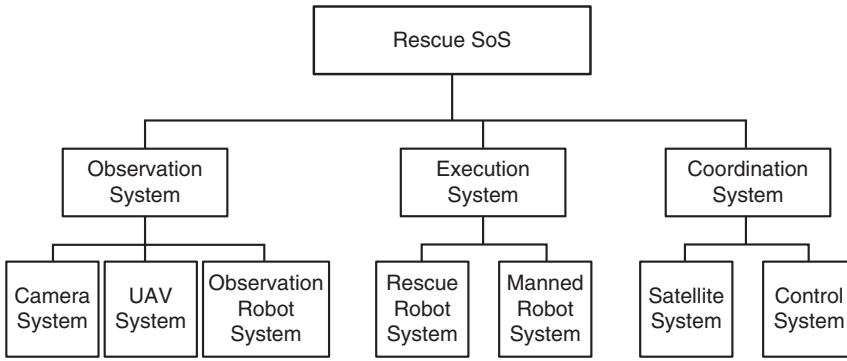


Figure 4.5 An SoS representation of the rescue system described in Chapter 2.

connected to multiple systems leading to decentralized and networked systems. These networked systems enable improved and enhanced communication and control. They allow the provision of functions that the contained systems would not be able to provide if they operated isolated from each other.

An example for an SoS is given by the rescue system that is described in Chapter 2. A sketch showing the decomposition of this example SoS into different systems is given in Figure 4.5. From the description of the system in Chapter 2, it is evident that the different systems in the SoS have extensive connectivity to exchange data with each other.

4.3 Definition of "Cyber-Physical System"

Miniaturization and increasing performance of electronics, a trend following the well-known Moore's Law [177], leads to comprehensive and automated control of technical systems and technical products. More and more of them, e.g. traffic systems, energy systems, production systems, automobiles, household appliances, or even lighting products, are equipped with more and more powerful control electronics. Moreover, such miniaturized and powerful electronics enable intensive integration of the digital world into our real world, i.e. an immersion of the physical world with the cyberspace, respectively. Previously, pure mechanical systems evolve via so-called embedded systems into contemporary cyber-physical systems (CPSs). Cars can be taken as an example for this evolution. The first motor coach by Carl Benz, a formerly pure mechanical system, has evolved into a system utilizing additionally electricity, e.g. for lighting or electric starter purposes. First electronic systems were added for engine control and safety functions, such as antiskid systems. Embedded systems with respective software have further

increased safety and comfort, e.g. for navigation and monitoring the vehicle environment. Finally, these various electronic systems with their digital data were merged into an integral "driver-vehicle-environment" system using information from the real world and the cyberspace, i.e. an example of a contemporary CPS.

The megatrend informatization becomes visible in multiple CPSs definitions. Many definitions utilize the term computing or computation instead of informatization. An example is the definition by Talcott who states that CPSs "integrate computing and communication with monitoring and/or control of entities in the physical world" [242]. Zhang describes CPSs as "next generation of engineered systems in which physical systems and cyber systems not only are converged, but also computing, communication, and control technologies are tightly integrated" [278]. According to Khaitan, CPSs are defined as systems "that offer integration of computation, networking, and physical processes" [141]. Song et al. [231] define CPSs as follows: "CPSs integrate cyber components (namely, sensing, computation, control, and networking) into physical components (namely, physical objects, infrastructure, and human users), connecting them to the Internet and to each other." Bondavalli et al. [32] define a CPS as "a system consisting of a computer system (the cyber system), a controlled object (a physical system) and possibly of interacting humans."

In contrast to an evolutionary development of SoS definitions, a timeline for CPS definitions is not observable. A reason might be that still today CPS definitions differ in the consideration of local and distributed systems. Some definitions consider self-contained systems with a well-defined system boundary [32, 141, 231], e.g. an aircraft or a modern production robot. Other definitions encompass multiple systems that constitute a CPS [242, 278], e.g. a smart energy grid or the air transport system. Consistency in these definitions exists for the functions computation, control, communication, and information exchange, but not all definitions consider possibly interacting humans.

Therefore, in this book, we use a combination of the definitions by Zhang [278] and Bondavalli et al. [32] as our CPS definition: CPSs are a next generation of engineered systems in which physical systems and cyber systems not only are converged, but also computing, communication, and control technologies are tightly integrated with possibly interacting humans.

4.4 Composition of a "Cyber-Physical System of Systems"

SoS and CPS are often denoted as smart systems if they can monitor their environment and adapt their behavior accordingly. A set of definitions [11, 230] considers self-contained systems as smart if these systems are able to monitor and influence

their environment. Therefore, systems require sensors, actuators, and their coordination by elements for signal processing and communication. This first set of definitions considering self-contained systems is applicable to embedded systems and if there is much communication with other systems, to CPSs.

Another group of definitions utilizes the attribute "smart" for large aggregated systems. Examples are smart cities [211], smart energy grids [205], and smart logistics [206] in the context of a Smart Logistics Framework. This group of definitions complies with SoS characteristics. In conclusion, smart systems do not constitute a system type that differs from SoS and CPSs. Instead, the attribute "smart" describes their ability to adapt to their environment and is applicable to SoS as well as CPSs when emphasizing their adaptive capabilities.

As introduced and described in Section 4.2 and 4.3, there might occur pure forms of SoS and CPS. However, due to their communality in communication which is nowadays based on ubiquitous connectivity and digital data and information transmission, merely all contemporary systems exhibit characteristics of both system types, i.e. SoS and CPS. Engell et al. [72] and Bondavalli et al. [32] are among the first authors who define Cyber-physical Systems of Systems (CPSoS). Guariniello et al. [97] applies SoS methodology to CPSs emphasizing that CPSs may show SoS characteristics, but does not classify them as CPSoS. This review shows that SoS research has a tradition that started in the 1950s when Boulding [35] and Simon [228] describe the need for better understanding of complex systems. At that time, interaction of their CS was not based on digital communication and microelectronics. The term CPS comes along with miniaturization and embedding of computing resources as well as ubiquitous and wireless connectivity. It is not striking that meanwhile properties of both system types are converging to form CPSoS. By expanding the concept from Figure 4.2 the relations of CPSoS, SoS, and CPSs are shown in Figure 4.6 using the newly introduced system types. Systems can contain other systems as system elements. CPSoS are a special type of SoS that contain at least two CPS which interact with each other in an SoS so that a CPSoS emerges.

Due to the digital revolution, a term which refers to the changes coming along with digital computing and communication technologies, there is an increased occurrence of CPSoS instead of pure CPSs or SoS. Formerly sheer functions of products or systems have evolved into services and capabilities of modern CPSoS-type products or systems. This evolution is often expressed by the trend of "Everything as a Service" [60].

Taking the example of cars from Section 4.3, new mobility services utilize multiple constituent systems, e.g., cars, user smartphones, and a software platform for booking and billing. New mobility services, e.g. carsharing or ridesharing services, can be provided by this mobility CPSoS, i.e. a service-oriented platform economy [86] with a novel kind of value network. Combined functions of the CS

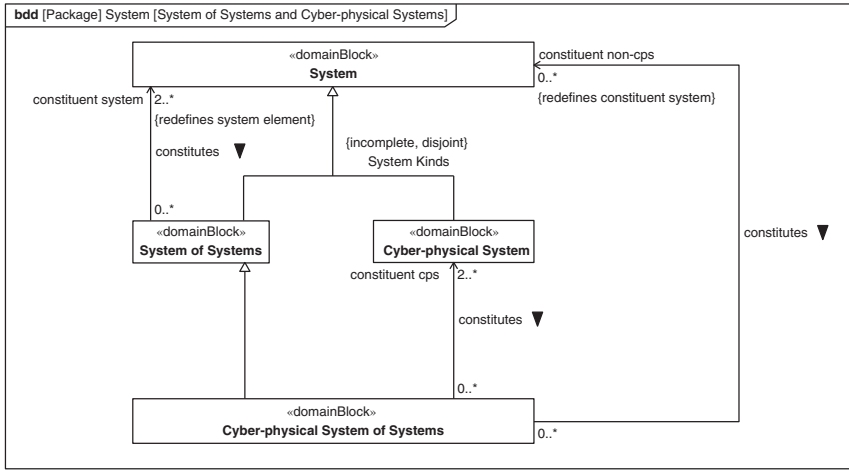


Figure 4.6 Cyber-physical Systems of Systems are special types of Systems of Systems that contain one or more CPS.

allow customers to detect cars available in their environment, to book these cars, to take the ride, and to do the payment and receive the billing for the ride. In such type of system, sensors for car localization and location-based services as well as biometric and electronic identification devices for locking and unlocking and for billing services are typical elements of CPSs including the interacting humans. SoS characteristics in such a system are represented by the connectivity and communication between the multiple CS, i.e. cars, user smartphones, and billing platforms.

The change to service-orientation affects traditional product development. Classical product engineering transforms into service engineering [39]. Instead of mainly isolated technical systems, product developers have to deal with platforms that integrate multiple systems into a CPSoS with many system interfaces and intensive data communication. This transition toward CPSoS development results in challenging engineering tasks and product life cycle management. When, e.g. a new car generation shall be integrated into a novel ridesharing platform it has to address both, the new and the legacy interfaces of the previous car generation to other CS, especially if there is a reasonable timeframe in that both generations should be operated in parallel [222, 256].

Along with the evolution of systems across SoS and/or CPSs toward up-to-date CPSoS, the extent and number of relations between system elements increases. Kurtz and Snowden [149] describe levels of complexity by simple, complicated, complex, and chaotic. While simple mechanical products may be simple from todays point of view, e.g. the bearing of a standard car, modern mechatronic systems and CPSs may be regarded as complicated due to multiple components.

Multiple connections of CS in a CPSoS are enhancing the interweaving CS to reach an even higher level of complexity which can be characterized as complex. Luhmann defines complexity in the legal system as "the sum total of the possibilities in experience and action, which, when actualized, amount to a meaningful structure" [160]. Cilliers [42] approaches complexity by collecting characteristics of complex systems. These characteristics encompass a large number of elements with dynamic and intensive interaction. Complex systems are usually open systems, interacting with their environment, and having a history that is responsible for their current behavior. CPSoS encompass all of these characteristics. Additional characteristics of complex systems suggested by Cilliers are the typical short-range interactions between system elements and the fact that system elements are ignorant of the behavior of the system as a whole. Please note that connectivity and digital communication in CPSoS allow long-range interaction and far-reaching data and information access. This is one of the strengths in CPSoS where information and knowledge transfer enables a better decision making within the autonomously acting CS, although these CS usually do not have a complete overview of the overall CPSoS.

Emergent behavior or emergent properties are typical characteristics of complex systems. They can appear, e.g. when a large number of system entities is interacting and operating together and exhibits novel unexpected functions or features. This example shows that the predictability and a deterministic development of CPSoS is not a simple and only directed task that may exceed human capabilities. Novel engineering approaches with software tool platforms for model-based development with product life cycle management try to cope with these enormous challenges [81]. The development of system architectures, as for example described in Chapter 3, is an important and fundamental task in Model-based Systems Engineering which can be applied to various types of systems, e.g. SoS, CPS, and CPSoS.

5

Definition of System Architecture

Defining "architecture" appears to be rather difficult considering the large number of existing definitions. These definitions share some commonality, though in detail differences become obvious. Definitions from different domains are debated controversially. For instance, one can observe enterprise architects refusing to accept the definition by ISO/IEC/IEEE 42010:2011 [114] with the rationale that an enterprise is no software intensive system. Software intensive systems are mentioned in the scope of the standard IEEE Std 1471-2000 [120], a predecessor of the before mentioned ISO/IEC/IEEE standard. This ISO/IEC/IEEE standard approaches the topic in a more generic way compared to its predecessor. And the definition for "architecture" in TOGAF® version 9.2 [245] meanwhile includes that of ISO/IEC/IEEE 42010:2011. Searching the online browsing platform of ISO [119] for the exact term "architecture" in the area "Terms & Definitions" results in 50+ hits. Considering terms comprising the word "architecture," such as "logical architecture," the search results in 190+ hits. The Software Engineering Institute (SEI) of the Carnegie Mellon University published a collection of definitions [274]. Though the latter lists definitions of "software architecture," it may serve as reference when defining "system architecture."

This book can hardly provide a global accepted definition for "architecture." Just adding another definition of architecture will not add benefit to the systems engineering community. Following the approach demonstrated in Chapter 4, we capture the widely accepted definitions of the standard ISO/IEC/IEEE 42010:2011 and integrate them in our model.

5.1 What Is Architecture? – Discussion of Some Existing Definitions

What makes a Gothic cathedral what it is? What is its architecture? What is the architecture of an elevator or a hearing aid system? What makes it usable, durable,

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

and beautiful? The last question traces to the roman architect Vitruvius Pollio (first century bce) who stated that architecture should be made to be durable, usable, and beautiful ("ratio firmitatis, utilitatis, venustatis" [203]). Architecture is a well established term related to buildings and other large constructions. Only in the recent decades, it became more frequently used in other engineering domains. Though, Vitruvius did not only mention buildings in his famous ten books "De Architectura Libri Decem." He considered architecture as a broad discipline following specific principles. Vitruvius identified three branches of architecture: buildings, time pieces, and machines or mechanics. He elaborated on planning of whole towns, buildings for various purposes, civil engineering topics such as water supplies, astronomy, sundials, and water clocks. And he considered machines as well. The considered machines were mainly to enable civil and military engineering.

The term "architecture" origins from "architect," the role that realizes the architecture of something. It derives from the term "architect" from Greek *arkhitektōn*, from *arkhi-* "chief" + *tektōn* "builder" [196].

The standard ISO/IEC/IEEE 42010:2011 defines "architecture" as fundamental concepts or properties of a system in its environment and the principles of its design and evolution. The standard defines these fundamental properties or concepts to be embodied in system elements and relationships. This definition is frequently referenced. A closer look the definition and descriptions in the standard reveals a couple of inconsistencies and ambiguities [266, 275]. The definition is ambiguous in relating architecture to concepts or properties. A concept stands for an idea whereas a property relates to a characteristic of the system-of-interest [196]. The informative annex A of the standard explains that the architecture definition was intended to consider two differing philosophies with no prejudice. Such differing philosophies may be related to the various styles of architecting, the action or process executed by architects. The guide "Introduction to System Architecting" [204] identifies four styles of architecting: authoritative, directive, coordinative, and supportive. For each of these styles, the use of architecture is a bit different. On one side, architecture is rather an idea to be realized, whereas on the other side, it is an explanation of constraints of the solution space.

A considerable number of definitions relate "architecture" to principles, e.g. [94]. Principles are accepted or professed rules [196] guiding the creation of the architecture. For instance, the architecture of a Gothic cathedral makes use of the principle that observers follow with their eyes the lines imposed with the design. This leads the observer to watch up toward the heaven. Maybe a requirement at that time was to make observers feel that the cathedral provides the relation between earth and heaven. Changing from an applied principle to another comes usually with a big effort and costs. The importance of principles in

architecture fit to the qualitative description that architecture includes everything that is difficult or expensive to change late in the development.

Emes et al. conclude in "Interpreting 'Systems Architecting'" [70], that a single-sentence expression of a root definition is hardly sufficient to cover the diverse viewpoints relating to many domains when defining "architect." This conclusion holds true for the definition of "system architecture" as well.

Practitioners consider architecting as a sequence of decisions. Decisions are mentioned in literature about architecting [102, 114, 226]. Nevertheless, definitions on "architecture" hardly show any relations to decisions. We will further elaborate on these aspects in the remaining sections of this chapter, following the same approach as for the definitions in Chapter 4.

5.2 Relations Between Concepts of "System," "Architecture," and "Architecture Description"

Before having a closer look to the definition of "architecture," we take a look on how the concepts of "system," "architecture," and "architecture description" relate to each other. A distinction between system and architecture seems to be widely accepted. A distinction between architecture and architecture description is not always made [104, 226]. The standard ISO/IEC/IEEE 42010:2011 distinguishes between the architecture and its description. Many literature follow this definition. But descriptions of the relations between the three concepts mentioned above are rare apart from those provided in ISO/IEC/IEEE 42010:2011. And the standard is partly inconsistent with these descriptions [266]. To make it even more complicate, the definition of architecture is very abstract and not easy usable in practice as we see hereinafter. Therefore, we introduce a fourth concept of "system architecture" which will be further detailed in this chapter.

Figure 5.1 depicts how the concepts of system, architecture, architecture description, and system architecture relate to each other. The detail definition of "System" is depicted in Figure 4.1, that of "Architecture" in Figure 5.2, "Architecture Description" appears in Figure 8.3, and "System Architecture" is depicted in Figure 5.6.

Architecture is abstract, an idea of something providing benefit to stakeholders [114]. An architect intends that the ultimate product follows these ideas. Therefore, the architect expresses ideas in appropriate architecture descriptions, identifying systems-of-interest. The development of architecture is a process with many steps. The architecture and related work products evolve during the architecting process. Architecting processes are not linear and include considerable potential for confusions. For proper identification of the architecting outcomes, we specialize the generic and abstract definition of architecture into more specific

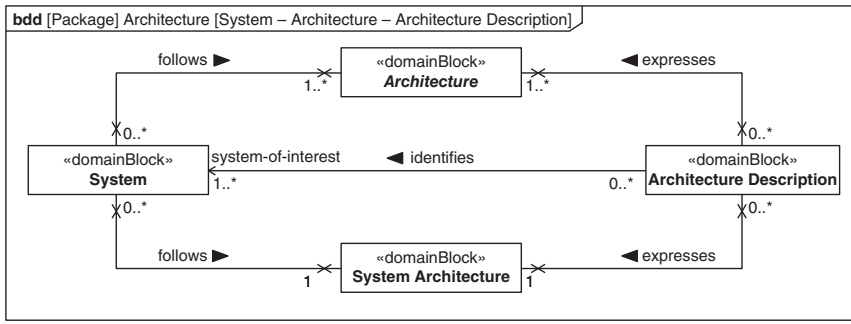


Figure 5.1 Relations between concepts of system, architecture, architecture description, and system architecture.

kinds of architectures. These specific kinds of architecture constitute the system architecture relevant for practitioners. Figure 5.6 comprises an upper part relating the abstract definition of architecture and a lower part relating the more tangible definition of system architecture.

The relation between system and architecture is depicted as association named "follows." The multiplicity on the system side of the association is [0..*]. Architecture can exist with no realized system. If an architecture is an idea defining only the necessary properties of a system, multiple systems can follow the same architecture. The multiplicity on the architecture side of the association is [1..*]. Multiple kinds of architecture can exist in parallel. Neither end of the association imposes properties to the related elements, hence neither end is navigable.

The relation between architecture description and architecture is depicted as association named "expresses" [114]. The multiplicity on the architecture description side of the association is [0..*]. Architecture can exist with no architecture description. And an architecture can be expressed in multiple architecture descriptions. The multiplicity on the architecture side of the association is [1..*]. Multiple architectures can be expressed in an architecture description. Neither end of the association imposes properties to the related elements, hence neither end is navigable.

The relation between system and system architecture is depicted as association named "follows." The multiplicity on the system side of the association is [0..*]. System architecture can exist with no realized system. Also, system architecture is still an idea defining only the necessary properties of a system. So, multiple systems can follow the same system architecture. The multiplicity on the system architecture side of the association is [1]. The system architecture is considered as a collection of all ideas necessary to realize a product. Neither end of the association imposes properties to the related elements, hence neither end is navigable.

The relation between architecture description and system architecture is depicted as association named "expresses" [114]. The multiplicity on the system architecture description side of the association is [0..*]. System architecture can exist with no architecture description. And a system architecture can be expressed in multiple architecture descriptions. The multiplicity on the system architecture side of the association is [1]. This is an idealized assumption. An architecture description about multiple architectures may be more confusing than beneficial. Neither end of the association imposes properties to the related elements, hence neither end is navigable.

The relation between architecture description and system is depicted as association named "identifies" [114]. The multiplicity on the architecture description side of the association is [0..*]. The system-of-interest can exist with no architecture description. If multiple architecture descriptions about an architecture can exist, also multiple architecture descriptions identifying a system exist. The multiplicity on the system side of the association is [1..*]. As architecture descriptions express architecture of systems-of-interest, it needs to identify at least one system. With multiple systems following the same architecture, the architecture description identifies multiple systems. Identifying systems in architecture descriptions does not impact the identified system. Therefore, the end on the system side of the association is not navigable.

5.3 Definition of "Architecture"

Architecture is abstract, an idea bridging from requirements to solutions. The idea is embodied in system elements building the system-of-interest. Architecture drives interactions between these system elements and between system elements and the system context. It comprises principles guiding the system's organization and evolution and includes architecture decisions made during the system's development and evolution. Figure 5.2 depicts architecture and its constituents. Additionally, it depicts a part of the definition of "System" (see Figure 4.1) because SysML imposes to depict association blocks with both, the related association as well as the related block.

The relations between architecture and its constituents are depicted with composite associations. The multiplicity on the architecture side is [0..1] for element that can exist without architecture and [1] for elements existing because of an architecture. A multiplicity of [0..1] applies for entities and interactions. We assume entities can naturally exist with no previous idea initiating their creation. Consequently, entities can interact with other entities. Such naturally existing entities can be on either side of the system boundary. Therefore, both system element interaction and context interaction can exist independent on

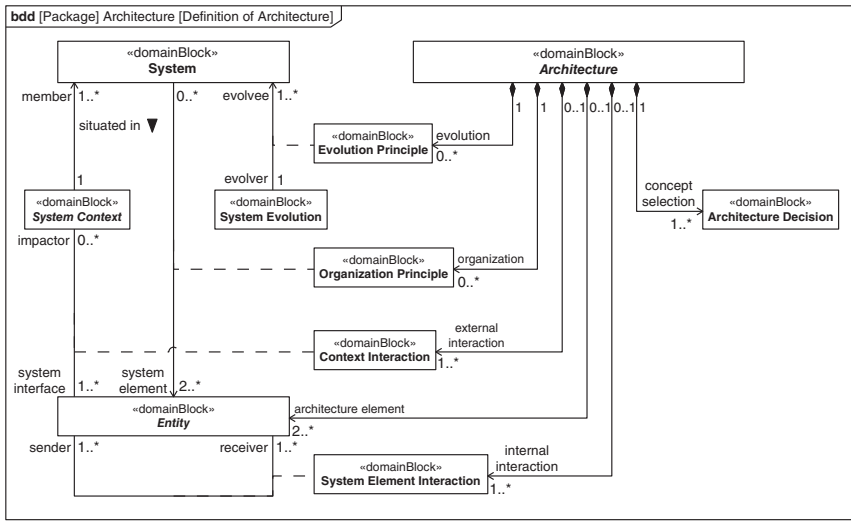


Figure 5.2 Definition of "Architecture."

architecture. Principles and architecture decisions exist because at least one architecture demanded them. Being a part of an architecture does not impact the properties of the constituents. Therefore, the composite associations are not navigable from the constituent sides.

Entities playing the role of system elements are part of the related architecture. In the context of defining architecture, the role of these entities is named "architecture element." Its multiplicity is [2..*], expressing that at least two entities are needed to form an architecture of a system. The upper bound is definition wise not limited.

5.3.1 Interactions

To interact means exchanging matter, energy, data, or combinations thereof [197]. Interactions stimulate behaviors of system elements contributing to emergence of system characteristics. Interactions may be desired or undesired by architects. They belong to the architecture in either case.

The role of system element interaction as constituent of architecture is named "internal interaction." Its multiplicity is [1..*]. A system comprises at least two system elements that interact. Consequently, the architecture needs to comprise of at least one internal interaction. The upper bound is definition wise not limited.

The role of context interaction as constituent of architecture is named "external interaction." Its multiplicity is [1..*]. An engineered system comprises at least one external interaction. We assume that engineered systems are not closed systems. The upper bound is definition wise not limited.

5.3.2 Principles

Principles justify at least some architecture decisions since they are regarded as fundamental truth or general scientific theorem [196]. Principles govern the organization of architecture elements and the evolution of the system-of-interest.

The role of organization principle as constituent of architecture is named "organization." Its multiplicity is [0..*]. Application of principles is not compulsory for the existence of architecture. The upper bound is definition wise not limited.

The role of evolution principle as constituent of architecture is named "evolution." Its multiplicity is [0..*], with the same justification as for organization principles.

5.3.3 Architecture Decisions

Developing and evolving an idea such as an architecture demands to decide between various concepts. Such architecture decisions form a relevant part of architecture [102].

The role of architecture decision as constituent of architecture is named "concept selection." Its multiplicity is [1..*]. We assume that always multiple concepts exist and consequently at least one architecture decision is needed to conclude to an architecture. The upper bound is definition wise not limited.

5.4 Functional and Physical Architecture

The definition of architecture as depicted in Figure 5.2 is of mediocre use for practitioners. In practical use are terms like "functional architecture," "physical architecture," "system architecture," "base architecture," "logical architecture," "product architecture," or "system design." Definitions for such terms vary, are conflicting, or are even omitted. This and remaining sections of this chapter elaborate on each of these terms regarding definition and relations.

Systematically bridging from requirements to solutions demands an evolutionary process. Such development process provides results evolving from vague to concrete and from general to detailed. Separating this wide field of architecture to more specific areas supports complexity management, understanding, and communication. An initial distinction is between functional and physical architectures.

A functional architecture defines the system-of-interest in terms of what it does. "What it does" refers to the input–output relation and not defining sequences of individual events realizing the transformation of an input to an output. Best practices in engineering suggest identifying functions before deciding on realizations using physical elements [56, 197, 226, 265]. Functional elements need

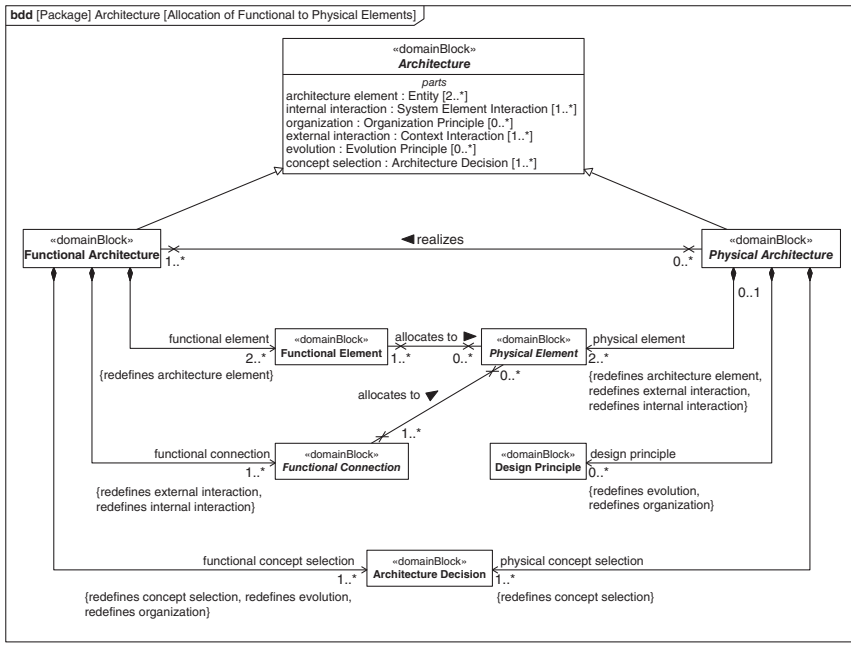


Figure 5.3 Architecture evolves though allocation of functional elements to physical elements.

to be allocated to physical elements. Consequently, decomposition of functional elements needs to proceed until an allocation to at least one physical element is feasible. A physical architecture defines how to realize functional elements identified in the functional architecture. That is, the physical architecture defines in what quality the system-of-interest realizes what it does. The decomposition of physical architectures needs to enable the univocal allocation of functional elements.

Figure 5.3 depicts the relations between functional and physical architecture. Both, functional and physical architectures, are specializations of architecture. The relation between functional and physical architecture is depicted as association named "realizes." The multiplicity on the functional architecture side of the association is [1..*]. A physical architecture realizes at least one functional architecture. The multiplicity on the physical architecture side of the association is [0..*]. A functional architecture can exist without physical realization as idea only. Multiple physical architectures can realize a functional architecture in various ways. Neither end of the association imposes properties to the related elements, hence neither end is navigable.

Functional architecture comprises functional elements, functional connections and architecture decisions. The relations between functional architecture and its constituents are depicted with composite associations. Functional elements

redefine the role and multiplicity of architecture elements. Functional connections are abstract and redefine the roles and multiplicities of internal and external interactions. Architecture decisions receive a role named "functional concept selection" with multiplicity of [1..*]. The role "functional concept selection" redefines the roles "concept selection," "evolution," and "organization." The multiplicity of architecture decision is [1..*]. To establish a functional architecture always requires to take decisions. Being a part of a functional architecture does not impact the properties of the constituents. Therefore, the composite associations are not navigable from the constituent sides.

Physical architecture comprises physical elements, design principles, and architecture decisions. Physical architecture is abstract and is specialized to logical architecture and product architecture as elaborated in Section 5.5. Physical element is abstract as well and specialized to logical element and product element in the taxonomy of physical architectures in Section 5.5. The relations between physical architecture and its constituents are depicted as composite associations. The multiplicity on the physical architecture side for physical elements is [0..1]. Physical elements can exist without being used by a physical architecture. Physical elements receive a role named "physical element" with multiplicity of [2..*]. Two or more physical elements are required to constitute a physical architecture. The role "physical element" redefines the roles "architecture element," "external interaction," and "internal interaction." Design principles redefine the roles "evolution," and "organization" with a multiplicity of [0..*]. Following design principles is not compulsory but can be expected in a structured development environment. The role "physical concept selection" redefines the role "concept selection." The multiplicity of architecture decision is [1..*]. To establish a physical architecture always requires to take decisions. Being a part of a physical architecture does not impact the properties of the constituents. Therefore, the composite associations are not navigable from the constituent side.

5.5 Taxonomy of Physical Architectures

An idea how to realize functions in the physical world can range from vague to concrete. Vague relates to solution principles only whereas concrete relates to acquirable solutions. These solutions exist in the physical world, and therefore, we consider them as physical architectures. For separation of concerns, we classify physical architecture into logical architecture, product architecture, base architecture, and layered architecture. Logical architecture is about solution principles, and product architecture is about principle solutions. Base architecture is about imposed solutions. Layered architecture is transversal to the other kinds of physical architectures and supports modularization. The application of layered architecture is elaborated in Section 11.5.

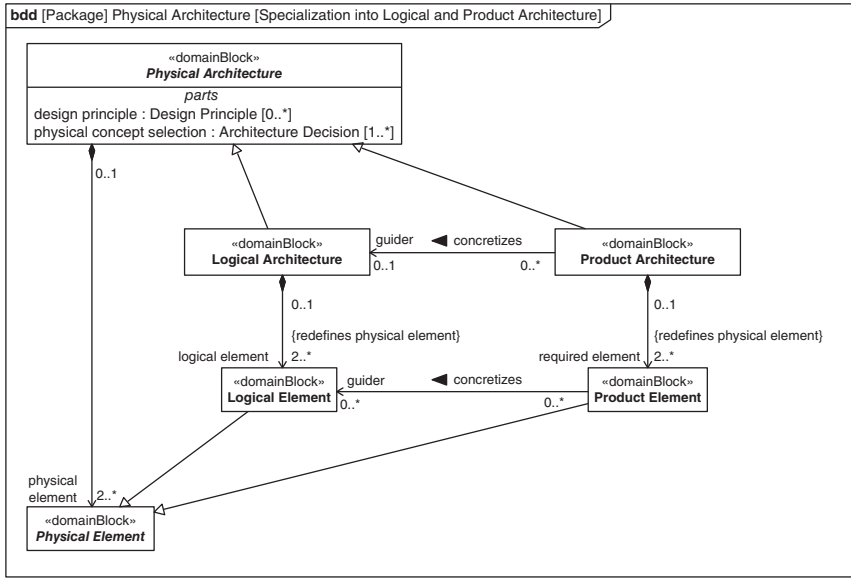


Figure 5.4 Specialization of physical architecture into logical architecture and product architecture separates solution principles from principle solutions.

Figure 5.4 depicts the specialization of physical architectures into logical architectures and product architectures. The specializations include redefinitions of physical elements into more specific logical elements and product elements. A logical element denotes classes of elements involved in realizing physical effects. Product elements concretize logical elements to an extend to make them acquirable.

5.5.1 Logical Architecture

The purpose of logical architecture is capturing domain knowledge. Logical architectures make use of solution principles. Solution principles relate to physical effects realizing functions [197]. In a specific domain multiple solution principles exist, each with advantages for specific applications. For instance, our example system, the virtual museum tour could be realized through application of the following principles (non-exhaustive list):

- offering a collection of pictures of an exhibition
- offering a collection of videos of various tours through an exhibition
- provide a robot with camera and life video stream with remote control by the customer

The various solution principles realize functions in different qualities and make use of different logical elements. Logical architectures need concretization to achieve an acquirable product. Concretization includes identification of required product elements that realize the demanded functions. Such product elements may exist and can be reused or the architect needs to specify them as far as their acquisition can be initiated.

5.5.2 Product Architecture

The purpose of product architecture is capturing product knowledge. Product architectures define principle solutions and identify the required product elements [197]. The required product elements need to be defined as concrete as required for acquisition. Acquisition includes development, purchasing, or reuse of existing elements. Product architectures shall not constrain the solution space for the required product elements more than necessary to realize the demanded functions. The final allocation of some engineering budgets may remain open and allocated during the system design. Consequently, some architecture decisions are postponed until more knowledge about realization of the product elements is available. Still, product architectures shall identify product elements contributing to these open engineering budgets.

5.5.3 Base Architecture

The purpose of base architecture is capturing knowledge of imposed solutions. Stakeholders impose solutions for various reasons. Such constrains can apply to various levels of concretization. We call such imposed solutions "base architectures." Base architectures are specializations of physical architectures. Imposed solutions can widely vary from imposing individual product elements to imposing substantial parts to the whole architecture. Base architectures have a relationship to requirements that is further elaborated in Chapter 10.

5.6 Architecture Landscape for Systems

This section completes the architecture landscape for systems as depicted in Figure 5.5. The idea about a system evolves from functional architecture to physical architectures. Physical architectures evolve from logical architectures to product architectures and may be constraint by base architectures. Definitions on system architecture, system design, and discipline-specific architecture and design complete the landscape to consider architecture as bridge between requirements and solutions.

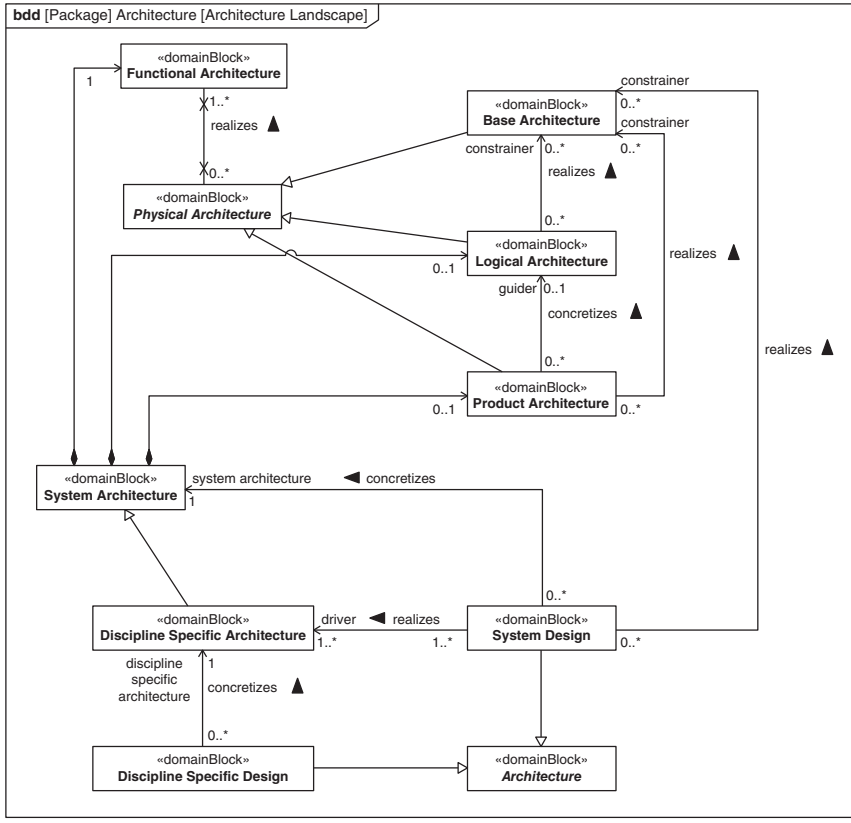


Figure 5.5 Architecture landscape for systems.

5.6.1 System Architecture

The purpose of system architecture is capturing system knowledge of what the system-of-interest does and how it can be realized to solve operational problems. System architecture is the composition of functional and physical architectures as defined in Section 5.4. Depending on the maturity of the system architecture, logical, or product architecture may not yet exist. A logical architecture may be skipped if capturing of domain knowledge is considered as not important.

Figure 5.6 depicts system architecture with its relations to specialized kinds of architecture, system, architecture description, and system design.

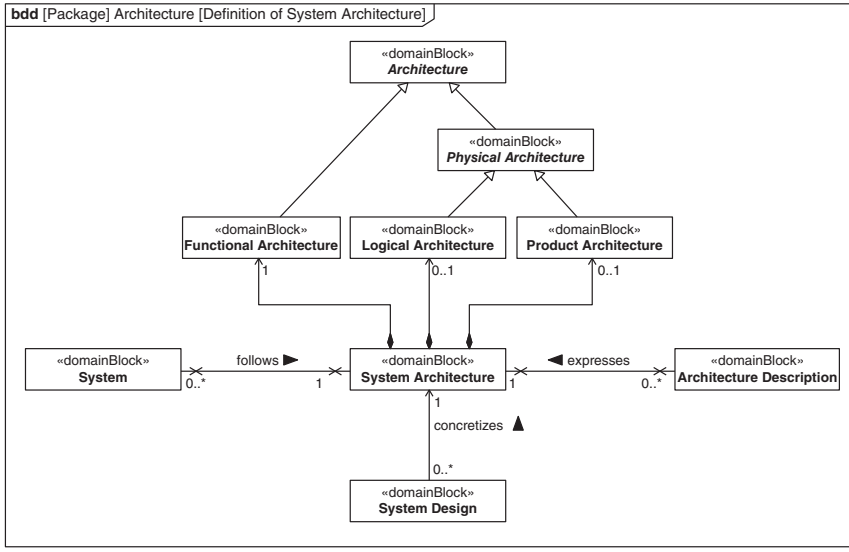


Figure 5.6 Definition of system architecture.

5.6.2 System Design

The purpose of system design is capturing implementation knowledge. That is, system designs concretize system architecture by solving implementation problems. Implementation is about realizing system elements [115, 265]. Consequently, these system elements need to be architected and designed prior to their realization. The concretization from system architectures to system designs is a trading between various solutions for system elements. It finalizes the allocation of engineering budgets and may demand reallocation of functional elements. This concretization is driven by the involved discipline-specific architectures.

From a high level perspective, architecture and design are synonyms. Both denote ideas on how to solve problems. Architecture is typically used with reference to more abstract solutions or when involving multiple disciplines. Whereas design is used with reference to concrete solutions or with single discipline involvement. In this respect, both system architecture and system design are kinds of architecture defining how to solve problems. System architecture (comprising functional and physical architecture) is about solving operational problems and system design (realizing discipline-specific architectures) is about solving implementation problems.

5.6.3 Discipline-Specific Architecture and Design

Multiple disciplines may contribute to a system architecture. Discipline [196] refers to both, specific domains or classical engineering disciplines including mechanics, electrics, and software. Each of these disciplines develops their specific architectures. Definition wise, such discipline specific architectures are specializations of system architecture. And they become concretizes to discipline-specific designs which solve their specific implementation problems.

6

Model-Based Systems Architecting

Digital engineering is a new upcoming discipline with a broader scope than model-based systems engineering (MBSE) [219]. It does not only include models but also other digital artifacts and covers the whole system life cycle, including the digital artifacts of all involved disciplines. Digital engineering is defined as "an integrated digital approach that uses authoritative sources of systems' data and models as a continuum across disciplines to support lifecycle activities from concept through disposal" [54]. Digital engineering will be the enabling technology to enable digital threads and digital twins, which in turn pave the way for the use of artificial intelligence technologies [169].

Model-based engineering (MBE) is a kind of digital engineering combining product lifecycle management (PLM) and MBSE approaches. The German chapter of INCOSE defines MBE "as the resulting concept of combining lifecycle spanning management of product data (PLM) and formal description of systems (MBSE)" [90]. The distinction between digital engineering and MBE is quite blurred, and both terms are also used as synonyms (for example, [181]). Nevertheless, we list both here since both are in circulation. In the following, we prefer using the term digital engineering.

The discipline MBSE is a kind of MBE focusing on systems engineering processes. INCOSE defines MBSE as "the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [122]. The most recent INCOSE Vision 2025 [123] mentions MBSE as the norm for systems engineering and system architecture as a crucial discipline for future successful systems engineering.

According to the definition of MBSE by INCOSE, we define model-based systems architecting:

> *Model-based systems architecting* (MBSA) is the formal application of modeling to support system architecture activities.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

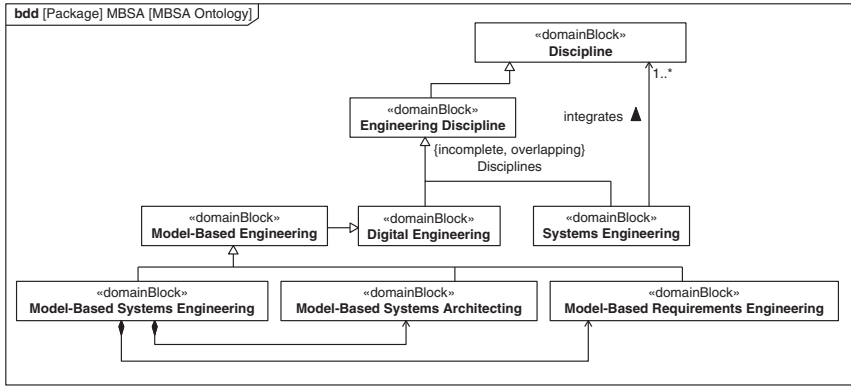


Figure 6.1 MBSA ontology.

A model-based system architecture is a system architecture that was obtained by or evolves via MBSA.

MBSA is a crucial part of MBSE as well as model-based requirements engineering and other MBSE subdisciplines. Figure 6.1 depicts the relationship between these terms.

Although the term *model* is essential, there is no common definition for the term *model* in the context of MBSE. Stachowiak defines in his book about general model theory three features of a model [233]:

- **Mapping**: A model is a mapping of something else.
- **Reduction**: A model only reflects parts of the original thing.
- **Pragmatic**: A model fulfills a specific function and is used in place of the original for this purpose.

The mapping and reduction features implies another often mentioned feature of models: abstraction. *Abstraction* is the process of reducing the information about a concept to the relevant parts for a particular purpose. See also Section 13.2.5 about the abstraction skill of system architects.

We second the features of Stachowiak and add some more to give a definition of how we understand and use the term *model*, respectively, the *system model* in the context of MBSE:

The *system model* (in the context of MBSE) is an abstraction of a real or to be realized system. The system model is characterized by the following properties:

- The entire system model may be composed out of multiple repositories, but from a user's point of view, it must behave like a single, consistent model.

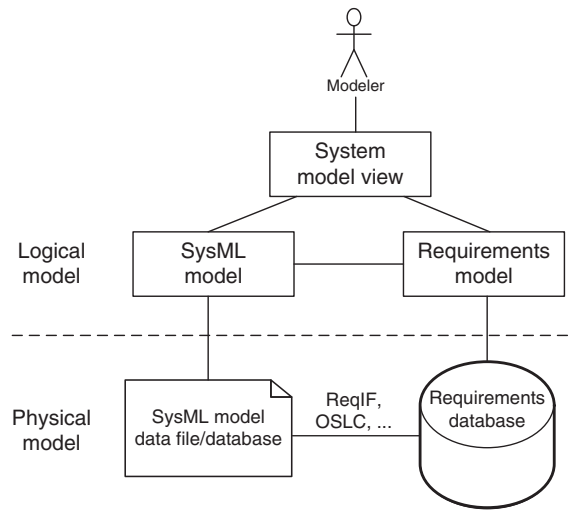
- The abstract syntax of the model language covers systems engineering concepts such as requirements, behavior, parts, and tests.
- The model enables different kinds of views.

The definition differentiates between the model and the repository. A *repository* is a manifestation of a model stored in a data storage, for example, a file or database. For example, a SysML model could be stored in a file and a requirements model in a database. These repositories are connected, for example, by using the ReqIF data exchange format [184]. The connection is, typically, discontinuous and does not exist all the time. Therefore, it is important to track the versions of the repositories to define a valid configuration. This is part of the configuration management.

From the user point of view, both repositories – the requirements database and the SysML model in a file – act like a single model. The user can navigate from the requirements to the architecture elements that satisfies those requirements and vice versa without recognizing that she crosses the border of the models (Figure 6.2).

We know that this is a challenging feature, and nowadays, typically, not fully implemented in current modeling tool landscapes. However, it is covered by many research projects like the FAS4M[1] project that closes the gap between system and CAD models or the CRYSTAL project.[2] That the user does not recognize that she

Figure 6.2 System model.



1 www.fas4m.de.
2 www.crystal-artemis.eu.

crosses the border of repositories is not a mandatory feature of a system model. It is sufficient that she can cross the borders.

The abstract syntax defines the model elements and the structure of the modeling language. It is not the same as the notation, which is called concrete syntax (Section A.2). Our definition of a system model requests that the abstract syntax of the model covers engineering concepts. For example, the abstract syntax of SysML includes model elements for requirements and system blocks, while the abstract syntax of a text document covers concepts like header or paragraph. For example, in a Hypertext Markup Language (HTML) document a header is enclosed by the keywords `<h1></h1>` and a paragraph by `<p></p>`: `<h1>This is a header</h1><p>This is a paragraph!</p>`. Therefore, a system description in a text document is not a system model but a text model. Although it could be a valuable view on a system model.

The system model must provide different views for the different stakeholder concerns. Typically, at least one view is a text document. Other views could be, for example, a set of SysML diagrams, or spreadsheets, or slide presentations. It is a strength of modeling that models provide multiple views of the same consistent information (single source of truth) to enable the communication between heterogeneous stakeholders.

Model-Based in the term MBSA means that the key artifacts of the system architecture are stored in a system model. In a document-based approach the key artifacts are stored in documents, and models are only used to contribute information to the documents, for example, SysML diagrams in a text document.

The term model-driven is also well known. There exists no common definition of the term and there are different meanings around about the difference of *model-based* vs. *model-driven*. Some say that *model-based* is a softer version of *model-driven*, which means the model is an important asset, but not the source. Other say that both terms are synonyms. We do not use the term *model-driven* in this book except for a short explanation of the next term MDA.

Another term often mentioned in the context of modeling is the OMG Model Driven Architecture™ (MDA™) [182]. MDA is a concept of the software engineering discipline where the model is the central asset of the development process. In a nutshell, MDA defines three model levels: computational-independent model (CIM), platform-independent model (PIM), and platform-specific model (PSM) (Figure 6.3). All three levels are a description of the complete system, but on different levels of abstraction.

CIM focusses on the domain and the language of the stakeholders. Typical artifacts of a CIM are use cases and texts. A PIM is already technology-oriented and more formal than the CIM but still independent of a specific software engineering platform like Microsoft.NET™ or Java2 Enterprise Edition (J2EE™). An example

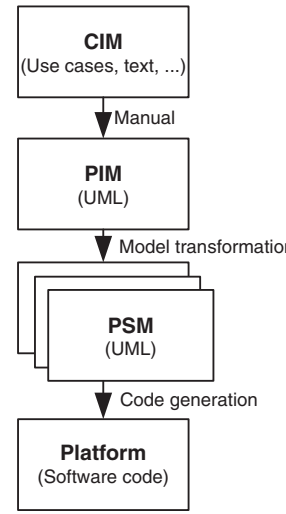


Figure 6.3 MDA levels.

of a PIM is the upcoming standard SysML v2 API & Services. The API and the services are defined by a PIM independent of any technology [239].

A PSM combines the specification of the PIM with platform-specific information. A PSM can be the result of a automatic model transformation from the PIM. For example, the SysML v2 API & Services standard contains a PSM based on REST/HTTP and a PSM based on "Open Services for Lifecycle Collaboration" (OSLC).

The basic notion of MDA is also interesting for MBSA. The concept of different model levels is partly reflected by separating the logical and product architecture, which could roughly be compared with a PIM, respectively, PSM (see Section 17.7 for a definition of logical and product architectures).

You can apply modeling at different levels of intensity. As an example, Figure 6.4 shows the SYSMOD intensity model [267, 269]. The three main levels define each a primary modeling goal.

Communication: The basic end of the modeling effort is to enable and improve the communication between the stakeholder of the project, including the development team. On that level, you have a strong focus on the view, i.e. the diagrams of the model. The model data itself is less important.

Traceability: The primary goal of modeling is the traceability between engineering artifacts. For instance, the connection from a physical block of the system to the set of requirements that the block satisfies. It is important to have a well-structured model to achieve valuable traceability.

Specification: This level is the real MBSE. The model is the master of the key artifacts of the system requirements and architecture.

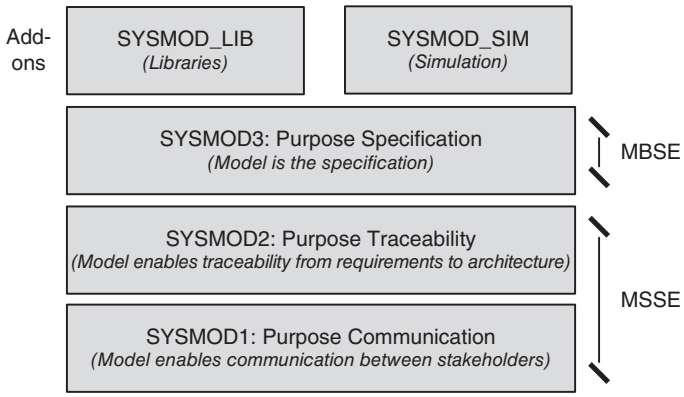


Figure 6.4 SYSMOD intensity model.

The approaches how to develop the content and the models are provided by MBSE methodologies. A methodology is a collection of related processes, methods, and tools [165]. The processes describe logical execution sequences of tasks, the methods describe how to perform the tasks, and the tools are instruments to enhance efficiency.

There are quite a few published MBSE methodologies, such as SYSMOD, OOSEM, OPM, ISE&PPOOA, or Arcadia. Typically, you do not use them out-of-the-box. Think of it more as a template upon which you develop your own customized methodology. How to derive a custom MBSE methodology from the needs of the project or organization is described, for example, by the SYSMOD Methodology Adoption Process (SMAP) [271].

This book does not describe a methodology, but it does describe methods and useful knowledge, patterns, and tools for system architects to work with MBSA. The approaches in this book do not contradict the known methodologies mentioned above but can be used for all of them. This book is about improving the craft skills of the system architect.

7

Model Governance

7.1 Overview

As shown in the Chapters 3 and 6, it can be advantageous to accompany the systems architecting activity with a model. However, the decision to model a system has consequences:

- The model needs to be created.
- The model needs to be maintained for a certain lifetime.
- People who depend on using information that is now in the model may need to be trained in the new way of retrieving their information, or alternatively, their known representation of information needs to be re-established based on the model.

The authors have observed modeling activities that failed, because the above consequences were not planned in advance, or – even worse – were not taken into account when defining the extent of the modeling activity. Too extensive modeling is what we also call *over-modeling*. Almost anything can be modeled, but you should only model those things which have a purpose in the value creation that is clearly confirmed by the related stakeholders.[1] Over-modeling usually leads to an explosion of cost, which cannot be justified by the much slower evolution of benefits. It is therefore extremely important to make the cost-benefit analysis before starting a modeling activity and to capture aspects like the intended scope, purpose, or lifetime of the model in order to have them for reference during the utilization stage of the model.

1 It is important to acknowledge that system architects themselves are important stakeholders of architecture modeling. When system architects become more successful in accomplishing their mission toward the business needs through a model, then this model is perfectly justified. In cases in which system architects model for themselves, it may however be important to obtain independent feedback about fulfillment of the model purpose, in order not to develop blind spots toward over-modeling.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

The word *model governance* shall denote the activity that comprises the following:

- The definition of a model creation and maintenance activity before it even starts, including boundaries of lifetime, purpose, scope, and other defining characteristics that are needed to plan the evolution and maintenance of the model.
- The follow-up on the actual model evolution in order to continually verify that the model stays within the defined boundaries.
- The initiation of corrective actions in case a model exceeds its boundaries.
- The continuous model maintenance and the initiation of exceptional intervention in case of unforeseen events (data corruption, software updates, …)
- The retirement of the model once its lifespan ends, and the proactive redirection of information requests that are addressed to the now retired model.

One may find similarities between the life cycle of the model and the standardized system lifecycle models we find, e.g. in ISO/IEC/IEEE 24748-1:2018 [116]. It would indeed be a feasible and correct idea to govern model development activities with the usual lifecycle management processes one also uses for managing system development. Nevertheless, this chapter will describe the model governance as a dedicated activity with dedicated terms and not necessarily with the terms of system lifecycle management. In other words: Even if it will theoretically be feasible to talk about model requirements, model architecture, etc., we leave it up to the reader to apply this kind of systems engineering rigor to modeling endeavors if needed (and if understood by the relevant "model stakeholders").

7.2 Model Governance in Practice

The modeling activity should start with the clarification of the model's purpose, its scope, its nature, and also by assigning responsibility and model user roles. One can think of a general questionnaire to answer as a first step of a modeling activity, for example:

- **Purpose of the model**: Which questions will the model help answer, or which problems shall it help solve?
- **Scope of the model**: Which items and phenomena of the real world shall the model describe or represent?
- **Nature of the model**: What kind of a model will it be? Will it be descriptive only? Can it be used for simulations? How detailed does it need to be in order to serve the above-mentioned purpose?

- **Owner of the model**: Which person or department will be responsible to monitor that the model complies to the above and to ensure that the model is maintained – or phased out and discontinued when no longer needed.
- **Contributors and users of the model**: Who are the people doing the modeling, and which other people will use the model? What is their professional background, and which model creation or interpretation competencies can be assumed they will have?
- **Relationships to other models**: Which other models like model libraries or discipline-specific models should be accessed by the model? And vice versa: which parts of the model should be made accessible to other models? Which parts should be provided in a model library?

We will see in Chapters 8 and 11 that the information to be retrieved from models is driven by stakeholder concerns. Looking at the architecture stakeholders and their concerns can help answering the above questions and help keep the goal in mind during the modeling activity [89]. We will see later in Section 11.1 how such thinking can be further formalized via the notion of a viewpoint.

Once the purpose, the scope, and the nature of the model are defined, they should be captured for later reference and for being followed up on. During the further modeling activity, it will become clear whether the initial definitions are sensible or whether they need to be adapted. At any point in time, the actual model should follow these definitions or their adapted versions and not deviate from them. Otherwise, the modeling activity is at risk of derailing:

- The modeling activity may suddenly take unforeseen dynamics and go into the wrong direction, making it impossible later to fulfill the original purpose of the model. This triggers the risk that the model will become more expensive than foreseen and in the worst case also completely useless.
- The model may suddenly scope-creep and hence cover items or phenomena that are already described elsewhere or are in the given context not sensible to describe at all.
- The model may become one of the wrong kind, e.g. a very expensive formal model that can be used for simulation – even though a very rough and unprecise abstraction of the system would have been sufficient. Note that a too detailed model may still solve the problem it was built for, but the maintenance cost of a detailed model is usually much higher than for a simple one.

Even if it is ensured that any modeling activity runs without derailing, there is more thought needed in the planning and maintenance of models: When considering how organizations work, we will typically find multiple departments that work

on very different problems and may each decide to apply modeling to solve some of them. Such modeling-friendly organizations should have control and overview of their modeling activities, in order to prevent the resulting models from becoming a jungle of hard-to-maintain or even unmaintained garbage. Keeping control and overview about the modeling activities in an organization is what we also call model governance.

It is not always easy to find the right modeling approach across an organization. As soon as multiple stakeholders have overlapping scope, there will be similar model elements in their models. It is a delicate trade-off to analyze whether common model elements should be maintained in joint libraries, or whether the risks resulting from their duplication can be accepted. In some cases, a common library will be difficult to obtain, for example, if multiple modeling languages coexists or if process and system models are used around the same solution space. Also, a common library creates dependencies between different models and the need to care about configuration management in handling the different model versions.

A good model governance approach will analyze such trade-offs, create transparency about the taken decisions and periodically revisit if the decisions are still valid in a potentially changed environment.

Once the above considerations have been made, a model or a set of models can be created. The model governance activity will now go on, in order to ensure that the models are still compliant with the boundaries resulting from their description of scope, purpose, etc. It will also ensure that model owners are still the right entities in the organization, even after reorganizations. Finally, it will ensure that model descriptions are updated if no longer appropriate and it will ensure that models are retired when no longer needed.

The retirement of a model comes with several questions that have to be addressed. Here are just some examples:

- How will stakeholders now obtain answers to the questions the model was supposed to answer?
- Will there be any references to retired models in other documents or models of the organization, and how will it be possible to handle these references?

Before starting a serious modeling activity, an organization should consider establishing model governance as described. This may create a hen-and-egg situation: Before having modeled a few aspects of the system-of-interest, an organization cannot judge whether the modeling is beneficial – but only if it is beneficial, it will be worth the effort to establish model governance. One way out of this dilemma may be to start with a dedicated pilot activity in modeling. The activity must have a defined scope and a defined end. It should be defined upfront if the resulting models can be thrown away at the end of the pilot activity. Based on the learnings made

during the pilot, the further modeling and initial model governance can then be established. The further modeling may best be accompanied by the following:

- "Cookbooks" or guidelines that ensure alignment about how to model and lead to consistent models.
- Model Life cycle Management: Model configuration management, Model Life cycle Information, etc.

It takes discipline to establish efficient model governance without too much overhead, but the long-term return-of-investment of a modeling activity should be reason enough to hold on to it.

8

Architecture Description

In Section 5.3, we define architecture as idea. Ideas are not observable and need communications and explanations to related stakeholders for successful system realizations. Architecture descriptions are tangible artifacts communicating and explaining ideas about systems-of-interest. Not only system realizations benefit from architecture descriptions. Observation of realized systems does not reveal each idea or justification leading to the system's existence [58, 266]. Therefore, architecture descriptions provide substantial benefit for maintenance and evolution of systems. As better architectures are expressed as better systems-of-interest, follow the ideas of their architects (see Figure 5.1).

Architecture descriptions shall document how selected solutions satisfy requirements or address concerns. Along with descriptions on how solving stakeholder problems, architecture descriptions document the inevitably taken or made architecture decisions including the related architecture rationales. That is, architecture descriptions express the how and the why of solutions.

Architecture starts to exist as initial solution hypothesis while analyzing and validating business and stakeholder requirements. Stakeholders may impose base architectures captured in concepts of operations (ConOps).[1] Frequently, such imposed architectures are not explicitly described. Often base architectures appear to the involved parties as obvious as not worth to spend effort expressing them in architecture descriptions nor documenting decisions why such solutions are imposed. Eliding architecture descriptions is not necessarily wrong and may be economically justified. Though, with no tangible architecture description,

1 We follow here the terminology of ISO/IEC/IEEE 29148:2018 [117] that makes a distinction between concept of operations (ConOps) and operational concept (OpsCon). The first refers to the intention on how to operate an enterprise or a company where the latter refers to the intention on how to operate the system-of-interest. Other literature does not make this distinction. E.g. version 3.2 of the INCOSE Systems Engineering Handbook [99] did explicitly refrain from such distinction.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

rationales and applied principles leading to specific solutions are probably not obvious to each relevant stakeholder. And a company may lose knowledge if the involved personnel change jobs. The absence of architecture descriptions may result in considering an architecture as unchangeable. Supposing unchangeable architecture imposes risks to miss more innovative or more economic solutions. Such risks exist not only for base architectures but for architecture in general.

8.1 Architecture Descriptions for Stakeholders

Architecture descriptions shall support communications between system architects and other stakeholders. Successful communication makes use of languages understood and accepted by each participant of that communication. Therefore, system architects need to identify stakeholders, why they are stakeholders, and which architecture description languages fit best for communication with these stakeholders. Section 23.1 elaborates on basic principles of communication that apply to architecture descriptions likewise.

Figure 8.1 explains why stakeholders exist. Stakeholders are individuals or organizations having interests in the considered system. Stakeholder's interests depend on concerns. Stakeholders have concerns because the considered system impacts processes in which stakeholders participate. Impacts to stakeholder processes by the system-of-interest happen during the whole system live cycle and can be intended or unintended. Figure 8.1 depicts stakeholder processes as an

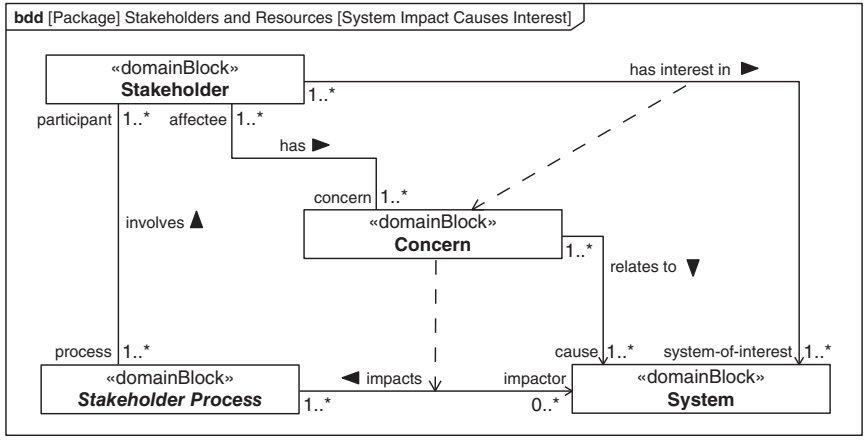


Figure 8.1 Impacts of any kind on processes participated by stakeholders cause concerns. Concerns result in interests in the impacting system.

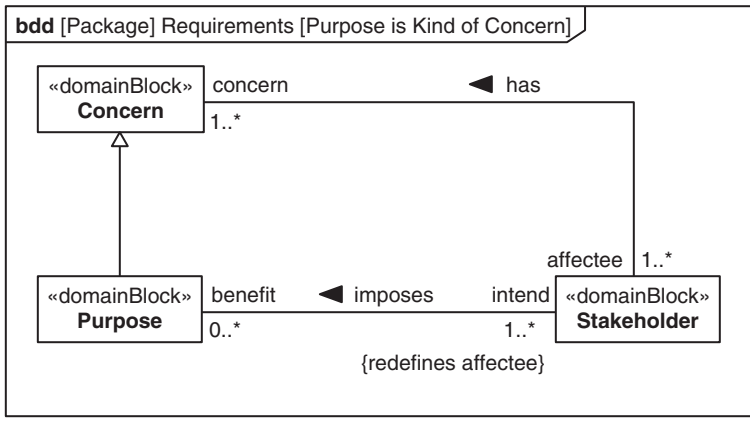


Figure 8.2 The purpose is a kind of concern.

abstract block representing any actions or operations participated by the related stakeholders.

A special kind of stakeholders are architecture stakeholders involved in the architecting process. Their interest in the system-of-interest extends to the system architecture. System architecture stakeholders are interested why and how the system-of-interest exists. Chapter 12 elaborates on typical system architecture stakeholders.

A special kind of concern is purpose [114] as depicted in Figure 8.2. A purpose explains why stakeholder benefit from using the system-of-interest (see Section 4.1.4). An engineered system starts to exist since its system characteristics permit achieving an initial purpose. The initial purpose justifies engineering efforts for the system as it provides sellable benefits for customers. System architects should be aware that stakeholders may impose other than the initial purpose to a system at any time during the system's life cycle.

System architects will analyze the identified stakeholders and their concerns to establish appropriate communications. Based on such analysis, system architects will select an fitting kind of presentation. This is a crucial decision. If stakeholders do not like the form of representation, they likely will not be in favor of the represented content.

Architects will optimize the effort spent into architecture descriptions in relation to the stakeholder's impact to the system's success. In most cases, architecture descriptions will not cover each concern nor provide documentation for each stakeholder. Concerns regarded as relevant for the system's success are elicited to requirements. Chapter 10 elaborates about analysis and definition of requirements and their tight relation with architecture.

8.2 Definition of "Architecture Description"

The definitions and explanations in this section build up on the concept of "architecture description" and related terms described in ISO/IEC/IEEE 42010:2011 [114]. These terms relate to the concepts of "system" and "architecture" as presented in Figure 5.1.

Architecture descriptions shall express architecture. Expressing architecture means communicating how selected solutions solve problems and why these solutions were selected. For such purpose, architecture descriptions comprise of four major architecture description elements[2]:

- architecture viewpoints defining how to communicate
- architecture views depicting the architecture
- architecture decisions tracing the evolution of architecture
- architecture rationales justifying architecture decisions

Ideally, an architecture description includes each of above-mentioned four architecture description elements. Economic reasoning may result in eliding some of them. In minimum, the architecture description shall identify:

- system-of-interest
- context of the system
- considered system elements
- addressed stakeholders
- covered requirements

The standard ISO/IEC/IEEE 42010:2011 does not mention requirements. It only considers concerns as drivers for architecting. From a systems Engineering, perspective formalized requirements elicitation is an important pillar toward the success of a system. We make a distinction between regarded and disregarded concerns. A requirements engineering process transforms regarded concerns into validated requirements (see Chapter 10). Disregarded concerns are considered as not important for the system's success. Still, disregarded concerns may be considered in architecture descriptions for communications with important stakeholders. Communications with important stakeholders may include justifications about disregarding concerns.

Figure 8.3 depicts the definition of "architecture description." As already described in Section 5.2, a system can exist with no architecture description. And more than one architecture descriptions can exist for a system. Both can

2 Architecture description elements are building blocks used to represent the how and the why architecture solves problems [114]. This definition is very broad and includes elements of various abstraction levels.

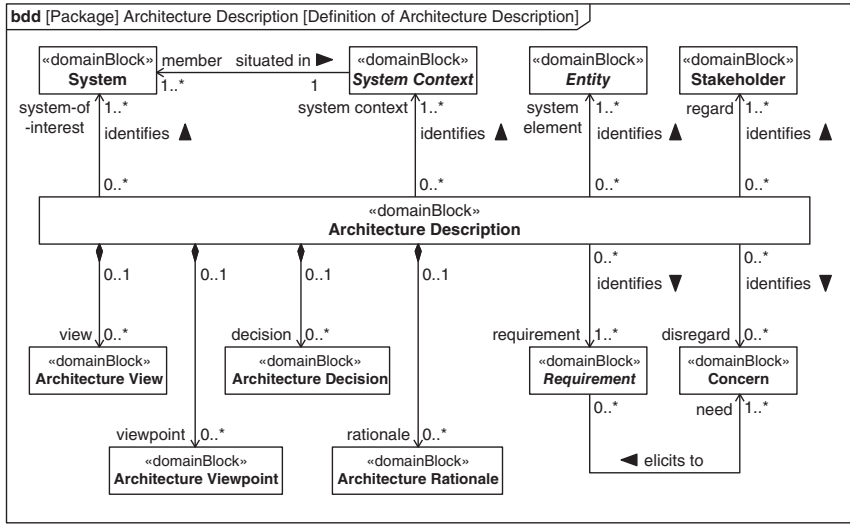


Figure 8.3 Definition of "Architecture Description."

be observed in practice. The latter causes some challenges to keep different architecture descriptions in sync, especially with a document-based approach. Maintaining data of the system's architecture in a model eases the generation of architecture descriptions as needed and enables to keep them synchronized with only limited additional effort.

Figure 8.3 defines the relation between architecture description and the four major architecture description elements (view, viewpoint, decision, and rationale) as composite association. Considering potential incomplete architecture descriptions, each of these architecture description elements has a multiplicity of [0..*]. And each of these architecture description elements can exist independent on an architecture description. Therefore, the multiplicity on the architecture description side is defined as [0..1].

The relations between the other architecture description elements (system, system context, system element, stakeholder, requirement, and concern) are depicted as associations and named "identifies." The multiplicity on the architecture description side of the association is each [0..*]. These architecture description elements can exist without architecture description and can be related to many architecture descriptions. The multiplicity on the system context side is [1..*]. Unlike the definition presented in Section 4.1.2, architecture descriptions may need to distinguish and identify multiple kinds of system context in parallel. The multiplicity on the sides of entity, stakeholder, and requirement is each [1..*]. An architecture description appears only appropriate if it identifies at least

one system element, one stakeholder, and one requirement. The multiplicity on the concern side of the association is [0..*]. Not each architecture description identifies disregarded concerns. The upper bound of the latter four multiplicities is definition wise not limited.

Each of above-mentioned associations are not navigable on the architecture description element side since architecture descriptions do not impact properties of these architecture description elements.

The relation between concern and requirement is depicted with an association named "elicits to." The multiplicity on the concern side of the association is [1..*]. Requirements are elicited from one or more concerns. The multiplicity on the requirement side of the association is [0..*]. Not each concern is regarded as relevant to be elicited to requirements, and multiple concerns can be elicited to a requirement. The association is not navigable from the concern side since requirements do not impact properties of concerns.

8.2.1 Architecture Viewpoints

Architecture viewpoints define how to communicate architecture. The communication makes use of visualization kinds defining how something shall be presented. Visualization kinds may be building blocks of architecture description languages. Architecture viewpoints define the communication only about a part of the architecture by framing requirements or concerns. Architecture viewpoints govern the related architecture views by defining how to present the framed part of the architecture. The relation between architecture viewpoints and stakeholders is indirect and can be derived from framed requirements or concerns and selected visualization kinds. Requirements and concerns relate to stakeholders. And stakeholders understand certain visualizations governed by the selected visualization kinds. Therefore, certain architecture viewpoints fit for communications with related stakeholders. System architects will need to evaluate which architecture viewpoints are necessary and sufficient for successful communications. Good practice demands documenting the rationales of the evaluation and selection of architecture viewpoints [89, 114]. Architecture frameworks support system architects in this regard by explicitly relate architecture viewpoints with stakeholders. Chapter 19 further elaborates on architecture frameworks.

Visualization kinds are conventions on how to visualize architecture description elements. ISO/IEC/IEEE 42010:2011 [114] designates such conventions for visualizations as "model kind." Examples of visualization kinds include:

- conventions on how to depict relations between system elements as N-square diagram
- conventions on how to depict traceability in natural languages in tables

- conventions on how to depict structure as block definition diagram and internal block diagram in SysML
- conventions on how to depict function as IDEF0 diagram

Architecture description languages are forms of expressions to describe architectures [114]. In its simplest form, an architecture description language makes use of a natural language. As any language, they can be formalized and specialized to specific aspects. Examples of such specialized architecture description languages include SysML or UML. Architecture description languages comprise of visualization kinds and correspondence rules [114]. Correspondence rules are kind of grammar of architecture description languages governing the relations between visualization kinds or between other architecture description elements.

Figure 8.4 depicts the definition of "architecture viewpoint." The relation between architecture viewpoint and requirement is depicted with an association named "frames." The multiplicity on the architecture viewpoint side of the association is [0..*]. Requirements can exist without architecture viewpoints and requirements can be framed by multiple architecture viewpoints. The multiplicity on the requirement side of the association is [1..*]. Architecture viewpoints frame one or more requirement. In a complete architecture description, each requirement is framed by at least one architecture viewpoint. The association is not navigable from the requirement side since architecture viewpoints do not impact properties of requirements.

The relation between architecture viewpoint and concern is depicted with an association named "frames." The multiplicity on the architecture viewpoint side of the association is [0..*]. Concerns can exist without architecture viewpoints, and concerns can be framed by multiple architecture viewpoints. The multiplicity on the concern side of the association is [0..*], and the role is defined as

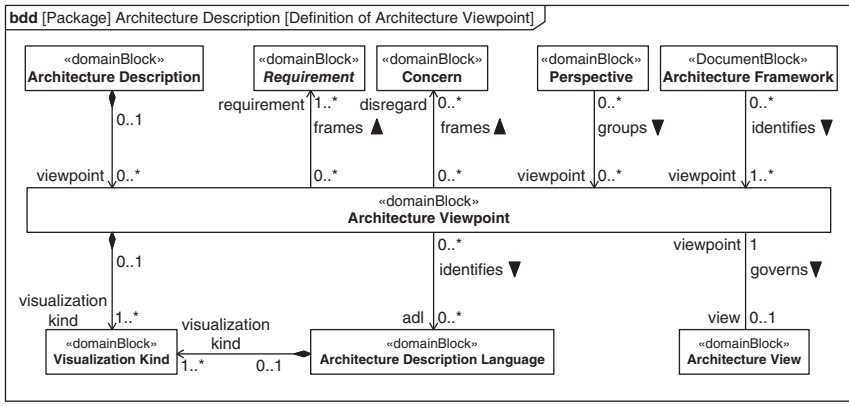


Figure 8.4 Definition of "Architecture Viewpoint."

"disregard." As mentioned above, regarded concerns are elicited to requirements. Communications may demand to include explanations on disregarded concerns. Consequently, architecture viewpoints can frame disregarded concerns. The association is not navigable from the concern side since architecture viewpoints do not impact properties of concerns.

The relation between architecture viewpoint and visualization kind is depicted with a composite association. The multiplicity on the architecture viewpoint side of the association is [0..1]. Visualization kinds can exist without architecture viewpoints. The multiplicity on the visualization kind side of the association is [1..*]. Architecture viewpoints comprise one or more visualization kinds. The association is not navigable from the visualization kind side since architecture viewpoints do not impact properties of visualization kinds.

The relation between architecture viewpoint and architecture description language is depicted with an association named "identifies." The multiplicity on the architecture viewpoint side of the association is [0..*]. Architecture description languages can exist without architecture viewpoints, e.g. before the first architecture viewpoint makes use of them. Architecture description languages can be identified by multiple architecture viewpoints. The multiplicity on the architecture description language side of the association is [0..*]. Architecture viewpoints may identify architecture description languages. An architecture viewpoint can be sufficiently defined using visualization kinds not belonging to an architecture description language. The association is not navigable from the architecture description language side since architecture viewpoints do not impact properties of architecture description languages.

The relation between architecture description language and visualization kind is depicted with a composite association. The multiplicity on the architecture description language side of the association is [0..1]. Visualization kinds can exist without architecture description languages. The multiplicity on the visualization kind side of the association is [1..*]. Architecture description languages comprise one or more visualization kinds. The association is not navigable from the visualization kind side since architecture description languages do not impact properties of visualization kinds.

The relation between architecture viewpoint and architecture view is depicted with an association named "governs." The multiplicity on the architecture viewpoint side of the association is [1]. An architecture view is governed by precisely one architecture viewpoint. The multiplicity on the architecture view side of the association is [0..1]. Architecture viewpoints can exist before creating the related architecture view.

The stringent definition of multiplicities between architecture viewpoint and architecture view has three consequences. It causes a considerable number of architecture views with some overlap. It demands a precise definition of the

architecture viewpoint. And the precise definition of architecture viewpoints permits automation to generate architecture views.

Managing large numbers of architecture viewpoints and subsequent architecture views demands for grouping for various purposes. The TRAK architecture framework [200] (see also Section 19.3.7) introduced the term "perspective." IEC 81346-1:2009 [105] defines "aspect" for analogous purposes, whereas ISO 15704:2019 [106] makes use of "perspective." In the context of this book, we define "perspective" as a way of perceiving architecture focusing to specific facets. Perspectives have some overlap with architecture frameworks (see Chapter 19). Both perspectives and architecture frameworks group various architecture viewpoints related to specific stakeholder or specific facets of architecture. In matrix representations of architecture frameworks, each column and each row can be regarded as a perspective. Perspectives are less formalistic and do not comprise explicit meta models or relate to specific development processes. Perspectives only serve for grouping architecture viewpoints for specific needs. Due to the stringent relation between architecture viewpoint and architecture view, perspectives indirectly group architecture views likewise. Chapter 11 elaborates on perspectives important for communication of system architecture.

The relation between perspective and architecture viewpoint is depicted with an association named "groups." The multiplicity on the perspective side of the association is [0..*]. Architecture viewpoints can exist without perspectives, and architecture viewpoints can be grouped into multiple perspectives. The multiplicity on the architecture viewpoint side of the association is [0..*]. Grouping to a perspective would demand two or more architecture viewpoints assuming a group has more than one member. But a perspective could be defined before defining the related viewpoints. The association is not navigable from the architecture viewpoint side since perspectives do not impact properties of architecture viewpoints.

The relation between architecture framework and architecture viewpoint is depicted with an association named "identifies." The multiplicity on the architecture framework side of the association is [0..*]. Architecture viewpoints can exist without architecture frameworks, and architecture viewpoints can be identified by multiple architecture frameworks. The multiplicity on the architecture viewpoint side of the association is [1..*]. Architecture frameworks identify one or more architecture viewpoints. The association is not navigable from the architecture viewpoint side since architecture frameworks do not impact properties of architecture viewpoints.

8.2.2 Architecture Views

Architecture views depict architecture. Governed by related architecture viewpoints, architecture views visualize how and why parts of the architecture

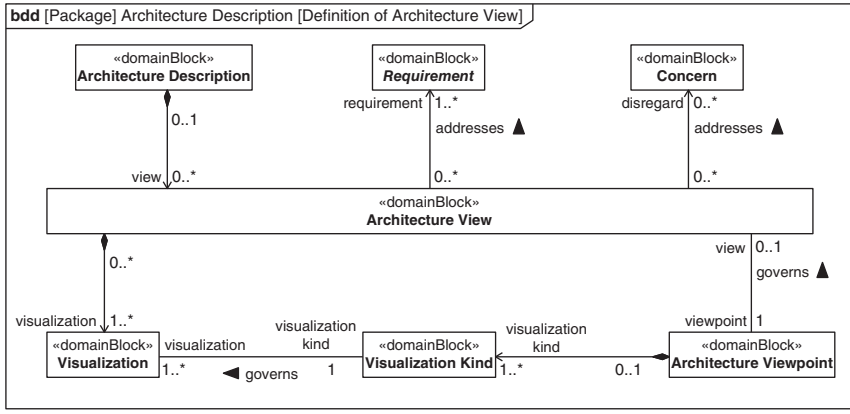


Figure 8.5 Definition of "Architecture View."

addresses requirements or concerns. Building blocks of these depictions are visualizations. Visualizations present architecture constituents and other architecture description elements according to the conventions of the governing visualization kind. ISO/IEC/IEEE 42010:2011 [114] designates these building blocks as "architecture models."

Figure 8.5 depicts the definition for "architecture view." The relation between architecture view and requirement is depicted with an association named "addresses." The multiplicity on the architecture view side of the association is [0..*]. Requirements can exist without architecture views, and requirements can be addressed by multiple architecture views. The multiplicity on the requirement side of the association is [1..*]. Architecture views address one or more requirements. The governing architecture viewpoint imposes which requirements an architecture view addresses. The association is not navigable from the requirement side since architecture views do not impact properties of requirements.

The relation between architecture view and concern is depicted with an association named "addresses." The multiplicity on the architecture view side of the association is [0..*]. Concerns can exist without architecture views, and concerns can be addressed by multiple architecture views. The multiplicity on the concern side of the association is [0..*], and the role is defined as "disregard." The governing architecture viewpoint imposes which disregarded concerns an architecture view addresses. The association is not navigable from the concern side since architecture views do not impact properties of concerns.

The relation between architecture view and visualization is depicted with a composite association. The multiplicity on the architecture view side of the association is [0..1]. Visualizations can exist without architecture views. The multiplicity on

the visualization side of the association is [1..*]. Architecture views comprise one or more visualizations. The association is not navigable from the visualization side since architecture views do not impact properties of visualizations.

The relation between visualization kind and visualization is depicted with an association named "governs." The multiplicity on the visualization kind side of the association is [1]. Visualizations are governed by precisely one visualization kind. The multiplicity on the visualization side of the association is [1..*]. A visualization kind can govern multiple visualizations.

8.2.3 Architecture Decisions

Architecture decisions are parts of both definitions, architecture (see Section 5.3.3) and architecture description. Documenting the why makes the development and evolution of architecture traceable. Architecture decisions are guided by requirements demanding new or changed systems. Requirements are elicited from concerns regarded as relevant for the system's success. Architecture decisions may raise new concerns which potentially become elicited to additional requirements. Architecture rationales justify architecture decisions. And architecture decisions affect architecture description elements by requiring or justifying their existence or change.

Documenting each architecture decision is hardly justifiable [114]. System architects need to judge the risks imposed by undocumented architecture decisions in collaboration with architecture stakeholders. Architecture decisions with high impact on stakeholder-values of the system-of-interest, and high impact on project cost, risks, and schedule need appropriate documentation. Dilemmas caused by conflicting requirements or trade studies with no clear favorite demand documentation of architecture decisions as well. Architecture decisions should comprise of a unique id, a decision statement documenting what the decision was, the date when the decision was made, the responsible, and in which role the decision was made.

Architecture decisions may depend on other architecture decisions. Such dependencies drive the sequence of making architecture decisions. System architects will need to evaluate the most appropriate sequence on deciding. Tackling interdependent architecture decisions easily becomes a complex process. A decision perspective and a decision road map support in managing such complexity aspects [226].

Figure 8.6 depicts the definition for "architecture decision." The relation between architecture decision and requirement is depicted with an association named "guides." The multiplicity on the architecture decision side of the association is [0..*]. Requirements can exist without architecture decisions, and multiple requirements can guide architecture decisions. The multiplicity on the requirement side of the association is [1..*]. Architecture decisions are guided by

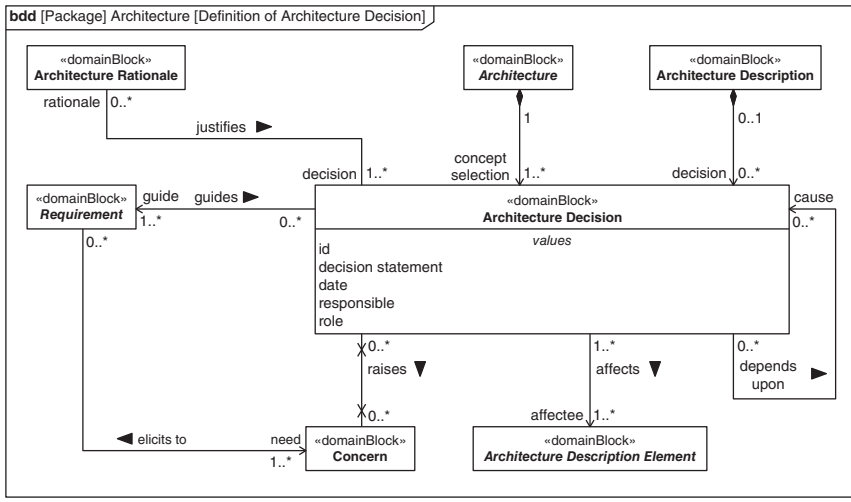


Figure 8.6 Definition of "Architecture Decision."

one or more requirements. The association is not navigable from the requirement side since architecture decisions do not impact properties of requirements.

The relation between architecture decision and concern is depicted with an association named "raises." The multiplicity on the architecture decision side of the association is [0..*]. Concerns can exist without architecture decisions, and multiple architecture decision can raise concerns. The multiplicity on the concern side of the association is [0..*]. Architecture decisions do not necessary raise concerns but architecture decisions can raise multiple concerns. The association is navigable from neither side since architecture decision do not impact properties of concerns and vice versa.

The self-relation between architecture decisions is depicted with an association named "depends on." The multiplicity on both sides of the association is [0..*]. The dependency between architecture decision can be of any multiplicity. The association is not navigable from the side that causes the dependency since a caused architecture decision does not impact properties of the causing architecture decision.

The relation between architecture decision and architecture description element is depicted with an association named "affects." The multiplicity on the architecture decision side of the association is [1..*]. One or more architecture decisions affect architecture description elements. The multiplicity on the architecture description element side of the association is [1..*]. Architecture decisions affect one or more architecture description element. The association is not navigable from the architecture description element side since architecture decisions do not impact properties of architecture description elements.

8.2.4 Architecture Rationales

Architecture rationales justify architecture decisions. Justifications make architecture decisions acceptable or at least understandable by providing evidences or arguments.

ISO/IEC/IEEE 42010:2011 [114] does not explicitly define architecture rationales. We consider hereinafter the following rationale kinds capturing evidences and arguments:

- Trade studies collect evidences to compare between alternatives.
- Reasoning builds a logical line of arguments considering constraints, assumptions, simulation results, risks, and opportunities.
- Application of principles follows accepted or professed rules. Four kinds of principles are relevant as architecture rationale: principles on design, interactions, organization, and evolution.

Comprehensive architecture rationales assist in dealing with liability issues. Architecture rationales can constitute domain knowledge. And architecture rationales provide evidence if revising of architecture decisions is appropriate.

ISO/IEC/IEEE 42010:2011 demands rationales for architecture viewpoints. The definition of "architecture rationales" presented here does not include rationales for architecture viewpoints. We consider rationales for architecture viewpoints as part of the architecture viewpoint documentation as mentioned in Section 8.2.1.

Figure 8.7 depicts the definition for "architecture rationale." Some kinds of architecture rationales are parts of both definitions, architecture (see Section 5.3.3) and architecture description. And some of the depicted blocks are not relevant for the definition of architecture rationale but need to appear per SysML specification on association blocks.

The relation between architecture rationale and architecture decision is depicted with an association named "justifies." The multiplicity on the architecture rationale side of the association is [0..*]. Architecture decisions can exist without architecture rationale, and multiple architecture rationales can justify architecture decisions. The multiplicity on the architecture decision side of the association is [1..*]. Architecture rationale will result in architecture decisions whether documented or not.

8.3 How to Get Architecture Descriptions?

8.3.1 Model-Based Vision

Pushing the print button and your modeling tool creates all the required documentation. This is the vision many system architect may have. Modeling tools,

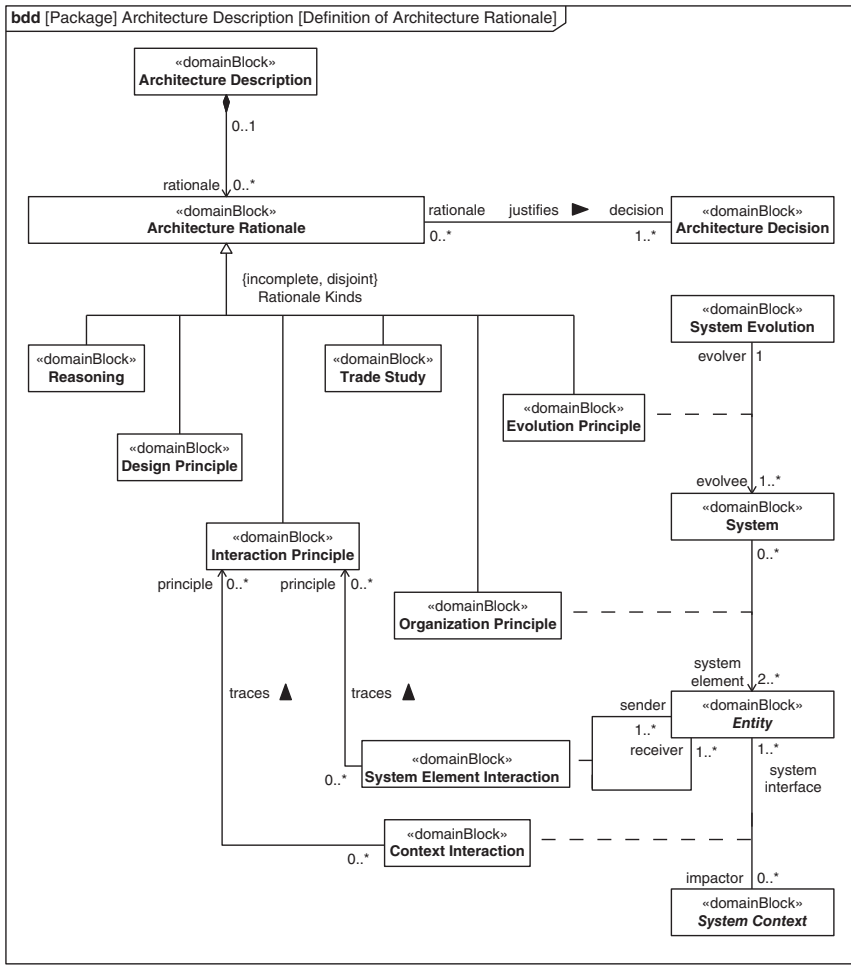


Figure 8.7 Definition of "Architecture Rationale."

with their heritage in software engineering do not always provide optimized capabilities to create descriptions of system architecture. Model-based approaches are very often limited to certain domains or disciplines. For many types of data exchange, documents are still required or even prescribed. We can expect that some of them will remain for a long time from now. A further challenge is that certain architecture views are used in documents being not considered as architecture descriptions in a first place. For instance, the standard on requirements engineering ISO/IEC/IEEE 29148:2018 [117] describes six kinds of documents. Five of them, the business, stakeholder, and system requirements specifications

(BRS, StRS, SyRS), as well as the concept of operations (ConOps), and the system operational concept (OpsCon), comprise a section with a view to the architecture of the system-of-interest. This example shows that architecture description or at least part of it need to be embedded into documents of other disciplines. Therefore, a comprehensive architecture model needs to support documentation in a rather extensive way. That is, it has to provide fragments of the architecture documentation in a form supporting the respective disciplines[3] in their documentation process.

An interesting approach in this regard had been demonstrated by the Telescope Modeling Challenges Team of the MBSE initiative by INCOSE and OMG [167]. They developed a plug-in to host documentation within a package of the architecture model. The structure in that package defines the structure of the document intended to create. It includes references to elements in the architecture model and provides containers for additional text. They called it Model-Based Document Generation. Though only rated as proof of concept, it shows a possible way in improving the documentation issue.

An additional approach includes automatic generation of views for architecture descriptions. Such views need generation on demand only. Modeling tools increasingly support automation. Therefore, we can expect that support for automatic view generation will become better as well. Still, practitioners may not be satisfied with today's solutions. And it is even not clear what solution should be provided, because needs for automation vary from industry to industry and from company to company. Literature on methodical approaches for automated documentation is sparse. In this context, it is worth mentioning the joint working group by the German and Swiss chapters of INCOSE evaluating the feasibility on generating architecture views from formally defined architecture viewpoints. That viewpoints working group fosters for a methodical approach in defining architecture viewpoints enabling generation of architecture views for the many stakeholders of MBSE. Initial results [89] show examples for some dedicated automation scenarios based on a modeling tool plug-in named Automated Instrument for Diagrams (AID) [20] which is maintained by the working group. We will briefly discuss this plug-in later in this book, in Section 11.11.3.

8.3.2 Forms and Templates

Model-based architecture description generations as mentioned in Section 8.3.1 are not yet widely established. In many cases, the expression of architecture demands manual document creation including the use of diagrams from models.

3 Here we refer to disciplines mainly acting on the same architecture level related to requirements engineering and verification.

Luckily, several forms and templates for architecture description exist. Often, however, they originate from software engineering and comprise sections that are not suitable for systems engineering. The working group on moderate complex systems associated to both, the German and Swiss chapters of INCOSE, presented such template at the "Tag des Systems Engineering," the German annual systems engineering conference 2014 [9]. Figure 8.8 depicts the headline Structure, and Table 8.1 provides explanations on the intended content for this template.

The target was to define a template with less than five pages, sufficient for moderate complex systems. The working group considered a number of sources and fostered for alignment with ISO/IEC/IEEE 42010:2011 [114]. The latter was not considered as regulatory requirement but as descriptive standard relying on experiences. The intention of this template is to get a single document useful for a limited number of stakeholders.

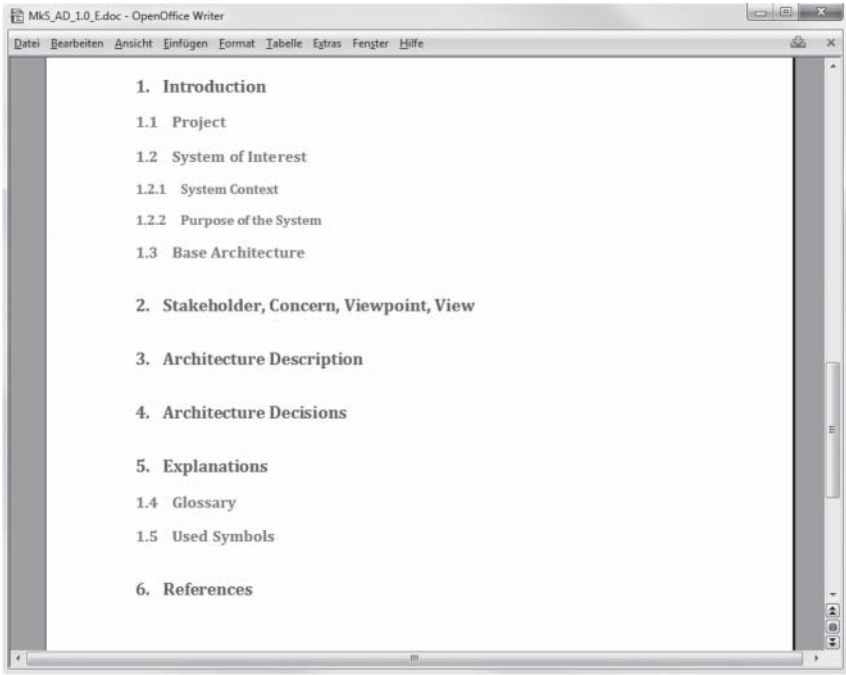


Figure 8.8 Headline structure of the Architecture Template 1.0 by the working group on moderately complex systems of Gesellschaft für Systems Engineering, e.V. (German chapter of INCOSE). Source: Reproduced with kind permission from the German and Swiss chapter of INCOSE.

Table 8.1 Explanation of content of architecture description template for moderate complex systems.

Heading	Intended content
1. Introduction	Short explanation on what the document is about.
1.1 Project	Description of the project in which the architecture description was created. This describes the context in which this work product had been created.
1.2 System-of-Interest	Introduction into the system-of-interest.
1.2.1 System Context	The system context comprises each entity interacting with the system in a not negligible way. This section shall identify these entities. Consequently, this section defines the system border. ISO/IEC/IEEE 42010:2011 designates the system context as environment.
1.2.2 Purpose of the System	The purpose defines why the system should be built. It may explain why it is beneficial for its users and customers are willing to pay for it. ISO/IEC/IEEE 42010:2011 defines purpose of a system as specialization of concern.
1.3 Base Architecture	Constraints imposed by stakeholders that limit the solution space. Such constraints may relate to the business case of the company that develops the system-of-interest. To impose the base architecture is an architecture decision. The related rationale is frequently linked to the business model or habits within companies.
2. Stakeholder, Concern, Viewpoint, View	A list of the considered stakeholders, their concerns and related viewpoints and views. It references the section of the document with the respective view. This is a kind of map of the document to direct each considered stakeholder to the sections of interest. This addresses sequential nature of a document that permits to optimize the sequence for only one viewpoint One of the simplest representations of these data could be a table. With more than only a hand full of stakeholders, other representations than tables should be considered.
3. Architecture Description	Description of structure and behavior of the system-of-interest in views as mentioned above. Each view is captured in a dedicated section. Obviously, this is the core of the document The structure within this chapter may serve to group-related views.

(Continued)

Table 8.1 (Continued)

Heading	Intended content
4. Architecture Decisions	Each architecture decision assigned with a unique identifier, for instance in form of a numbered list. Each decision should be linked to the name of persons that did take the decision and their roles at the date of the decision as well as to a comprehensive rationale for that decision.
5. Explanations	Description of vocabulary specifically used in this document. This refers to the used architecture description languages and could also comprise references to the related glossaries or standards.
5.1 Glossary	For natural languages, a list of terms with related explanations that are important to understand the content of the document.
5.2 Used Symbols	For graphical languages, a list of used symbols with the related explanations.
6. References	A list of referenced artifacts.

Another form was presented at EMEASEC 2014 by Beasley et al. [26]. They proposed a structure in analogy to the structure of well-established requirements documents. This approach supports readers that previously read the requirements document and eases to follow the transformation from requirements to architecture.

9

Architecture Patterns and Principles

Did you know that you use patterns every day? It is a powerful mechanism and much more powerful when explicit and known. A pattern describes a proven solution for a problem. Every experienced engineer has a huge set of patterns. Think about your daily work. If you have to solve a problem you have solved successfully before, you will remind your solution and apply it again. If you describe your solution in a general and reusable way, you have made your experience explicit, persistent, and reusable for everyone. Voilà! You have found a pattern.

Architecting is not a task that can be done simply with checklists and predefined processes. It is strongly based on the experience and talent of the system architects. Patterns are a good tool to make the implicit knowledge of the system architect explicit. Nevertheless, checklists and processes still support the architect and are also in one drawer of her toolbox.

Patterns and principles are documented best practices and proven experiences of system architects. It is recommendable to consider them, and you need very good reasons to work contrary.

You can't invent a pattern. You can find and describe a pattern. One of the first who did that and triggers the pattern community with his work "A pattern language" was C. Alexander [14]. His book is not about software or systems engineering patterns, but about architecture patterns for buildings. Of course, manifesting best practices by writing them down works for every discipline.

Principles are similar to patterns. Architectural principles describe guidelines that, in general experience, lead to good architecture when followed, while patterns describe basic working solutions for given problems.

This chapter lists architecture patterns and principles that we find useful and which we reference from other chapters in this book. The list is not complete, and you should identify your personal set for your architecture toolbox. For more information about patterns and systems engineering, we recommend the paper "Applying the Concept of Patterns to Systems Architecture" from Robert J. Cloutier and Dinesh Verma [44]. For more information about system

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

architecture patterns, we recommend the book "System Architecture – Strategy and Product Development for Complex Systems" from Bruce Cameron et al. [50].

9.1 The SYSMOD Zigzag Pattern

The pattern is described as part of the SYSMOD methodology in [267, 271]. It describes the relationship between requirements and architectures on different abstraction levels.

Requirements specify what the system should do and the system architecture and design how the system satisfies the requirements. This rule seems to be easy. However, on the second view, it will turn to be a challenging question. What is part of the architecture, and what is part of the requirements?

Let's assume you have absolutely solution-free requirements (although most of those requirements are not viable in practice). Now, you derive a system architecture that satisfies the requirements, and you get the typical what/how-pair. For example, consider the requirements of guided virtual live tour through a real museum or company. You derive a system architecture with mobile tour robots that satisfies the given requirements. That solution leads to new requirements that contain aspects of the solution, e.g. requirements about collision detection and avoidance of the tour robots with things or people in the building. You won't need those requirements when you have derived a system architecture, for example, without moving robots, but with a static camera system.

The requirements depend on a given architecture and are not free of any solution. They are on another abstraction level and solution-free from the viewpoint of that level, but they contain solution aspects of the previous level.

Again you derive a solution from the requirements, for example, a camera-based system for collision detection. That solution leads to new requirements and so on (Figure 9.1). All in all, the logical steps represent a zigzag pattern. "Zig" – from requirements to a system architecture – and "Zag" back from the architecture to the what-side of the derived requirements. Then again, a "Zig" from the derived requirements to the architecture, and so on.

The zigzag pattern also shows the relationship between the different requirement specifications and architecture kinds (Figure 9.2). The "Stakeholder Requirements Specification" is often the top-level specification in a project. It specifies the requirements of the stakeholders. Next, a system architecture that satisfies the requirements is derived. It is a logical architecture that specifies the technical concepts but omits the technical details. The functional architecture for systems (FAS) method can be used to create first a functional architecture and then derive the logical architecture (see Chapter 17 for the FAS method).

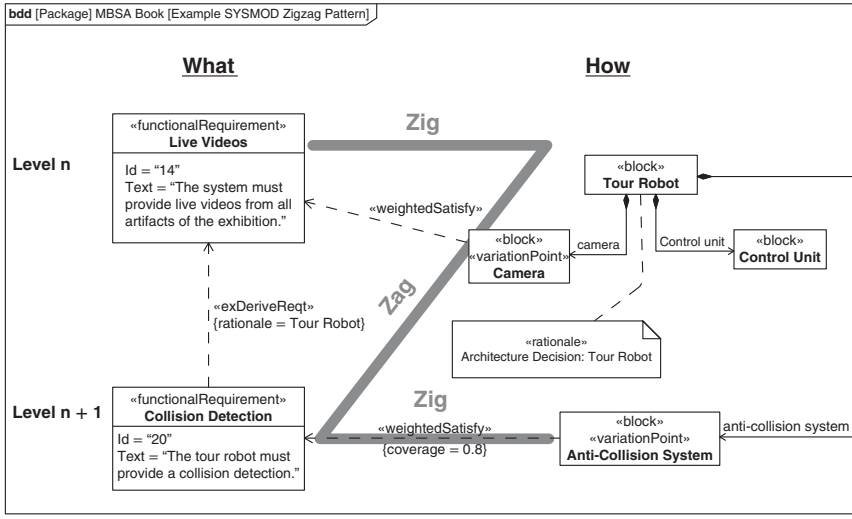


Figure 9.1 The SYSMOD zigzag pattern.

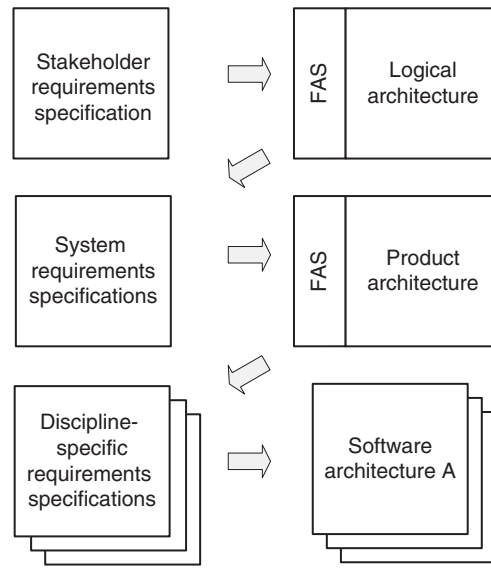


Figure 9.2 Relationship between requirements and architecture kinds.

The logical architecture is the base for the "System Requirements Specification." While the stakeholder requirements specification defines what the system should do from the stakeholders' perspective, the system requirements specification defines how the system should satisfy the stakeholder requirements from the engineers' perspective.

Next, the product architecture is derived from the system requirements specification. Again, a functional architecture can be elaborated in between. The product architecture satisfies the system requirements and includes all details necessary on the system level to build the system.

On the next level, you can derive the discipline-specific requirements for the software, mechanical, electrical, etc. assemblies. The functional architecture exists on each level as another architecture kind.

Probably that reminds you of the V-Model.[1] The zigzag pattern covers the left branch of the V-Model, as shown in Figure 9.3.

A common problem in practice is a strong focus on the left side of Figure 9.2. Stakeholders, principals, and others communicate and discuss mainly the requirements and not the architectures behind them. The requirements and their architecture one level above – the so-called base architecture – should be treated as one artifact in the engineering process. See also Section 9.2 for more information about base architectures. There are also projects with a strong focus on the right side, which means projects where the requirements are more in the background and the development is driven by the technology. In our opinion, the best option is as always to steer a middle course between the fairway buoys requirements and base architecture to the left and to the right.

The zigzag pattern is independent of SysML or any other requirements and architecture modeling languages or methodologies. It is a generic pattern that describes the relationship between requirements and architectural elements across abstraction levels. A similar approach is the function-mean tree developed by Andreasen [19] and the axiomatic design [236]. Another one is the Systems

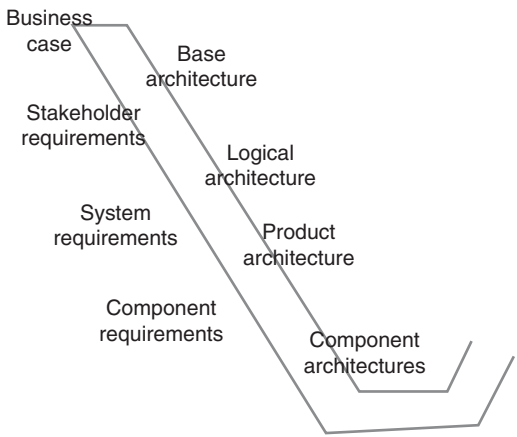


Figure 9.3 Zigzag pattern in the V-Model.

1 Readers not familiar with the V model can find its description and a short summary of its history in Appendix B.

Engineering Sandwich described by Jeremy Dick and Jonathon Chard [55]. All of them describe the co-evolution between requirements and solutions.

If you want to model the zigzag pattern explicitly with SysML, you need the following relationships and model structures: According to the model structure template given in Section 9.9, the system requirements are located in a top-level package called "<system>_Requirements." Typically, the package owns subpackages to further organize the requirements. The architecture elements that satisfies the requirements are located in a package "<system>_LogicalArchitecture" or "<system>_ProductArchitecture." Again, this package has further subpackages to organize the architecture. The elements of the architecture have satisfy relationships to the appropriate requirements. This structure models one level of the zigzag pattern (Figure 9.4).

On the next level, the requirements have derived relationships to one or more requirements from the upper level. They are based on architecture decisions made in the architecture. The SysML derive relationship itself has no property to store the rationale why and how the requirement was derived. We present two options to relate the rationale with the architecture elements to the derive relationship:

1. SysML provides the rationale element. A special comment to document the rationale why something was modeled like it is. Attach a rationale to the derive relationship and model anchors (dashed lines) from the rationale to the architecture elements (Figure 9.5).
2. Introduce a new stereotype for an extended derive relationship that specializes the SysML stereotype «deriveReqt» and adds a property to store the information about the architectural elements that lead to the derived requirements (Figure 9.6 and 9.7).

The best option depends on your needs. The first one has the most explicit visualization of the relationships. The second option could easily be accessed by tools to analyze the relationship.

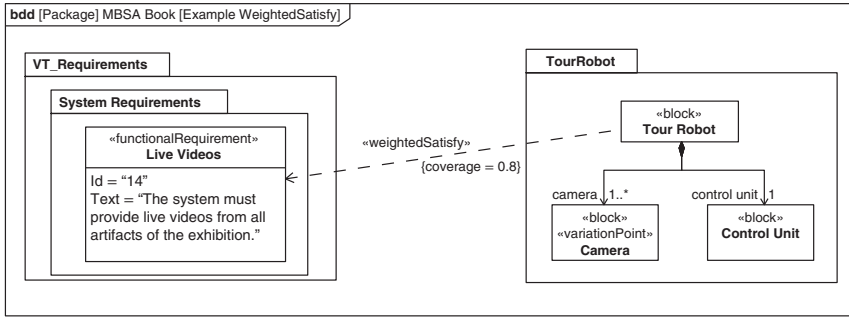


Figure 9.4 One zigzag pattern level in the model.

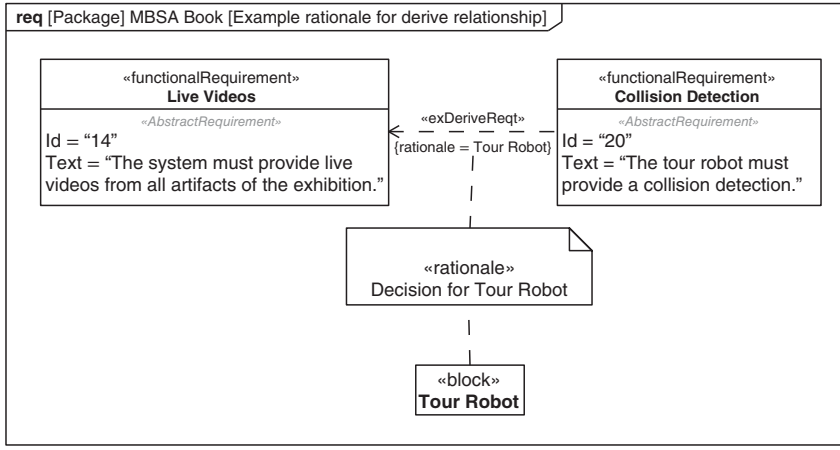


Figure 9.5 SysML rationale element to document derive relationship.

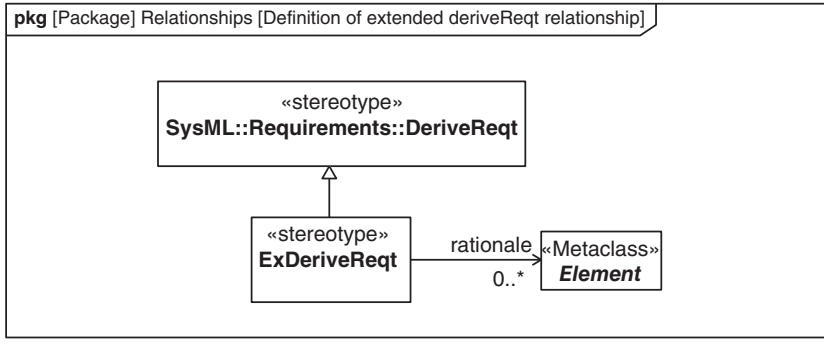


Figure 9.6 Definition of extended derive relationship.

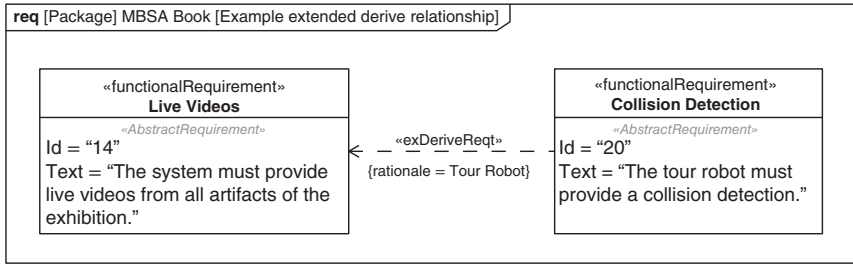


Figure 9.7 Example of the extended derive relationship.

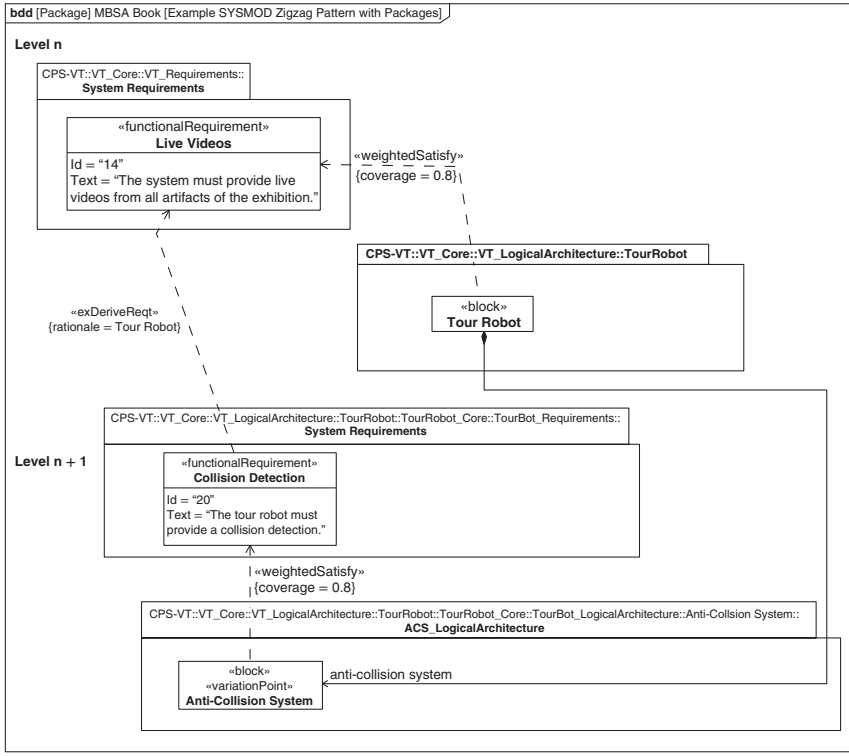


Figure 9.8 Model structure and relationships of the zigzag pattern.

Figure 9.8 shows a composition relationship from the tour robot block to the anti-collision system. That relationship crosses the border of the zigzag levels in the wrong direction. The lower level must depend on the upper level. Here the upper level depends on the lower level. In the strict sense, that is not allowed. However, as mentioned earlier, it is not necessary to strictly model the zigzag pattern. It costs effort to strictly separate the levels, and the effort should only be spent if you gain the benefit. To strictly separate the elements of the zigzag levels, you can introduce a specialized version of the tour robot as shown in Figure 9.9.

The zigzag pattern has value in communication and modeling. Even if you don't model the pattern structure explicitly, it demonstrates the relationship between requirements and the system architecture and the abstraction levels. It gives you a vocabulary to structure discussions about requirements and to conduct them in a constructive spirit. Equipped with the correct vocabulary, you can, for example, express the decision which levels of requirements or architecture to cover with a SysML model and which ones to omit during the modeling.

If you model the pattern explicitly, it provides traceability paths from the system architecture to the requirements over many abstraction levels. It clearly shows the separation of requirements engineering and systems architecting and at the same

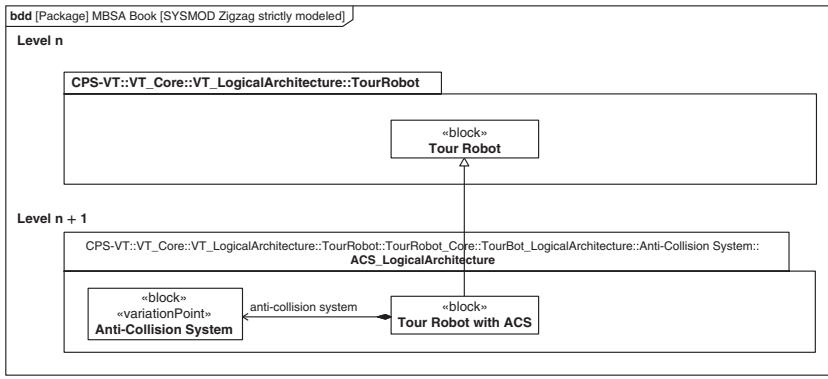


Figure 9.9 Strictly separated zigzag levels.

time shows the need for a close collaboration between the requirements engineers and the system architects, which will be discussed in detail in Section 12.2.

9.2 The Base Architecture

Your stakeholder or system requirements, typically, do not start at the very top of the zigzag pattern levels (Section 9.1), that means, they already include some technical decisions. An aviation company has requirements about planes and an automotive company about cars, and not about transportation systems in general. Most projects start on the brownfield, which means they build on existing systems.

Some of the architecture decisions that are already set with the start of the project are obvious, some are implicit. They are one of the causes why requirements are a sore spot of many projects. The knowledge and assumptions about implicit architecture decisions are different between the project members.

The zigzag pattern shows that the architecture decisions that are the input of the engineering project are described in the architecture one level above. From the viewpoint of the requirements that architecture is called the base architecture. Figure 9.10 shows an extract of the base architecture of the Virtual Tour system.

The virtual tour base architecture clearly specifies that the system receives power from the building's electrical installation, will have robots with wheels, an electrical engine and cameras, that are controlled by a virtual tour server. The top-level requirements of the system depend on this base architecture and directly use those elements. They are only solution-free according to their level in the zigzag pattern. For example, we have top-level requirements "Robot Speed" and "Robot Mass" that are based on the decision to use robots. We wouldn't have those requirements if we decided for a static camera system. Figure 9.11 shows the dependency of the requirements to the base architecture.

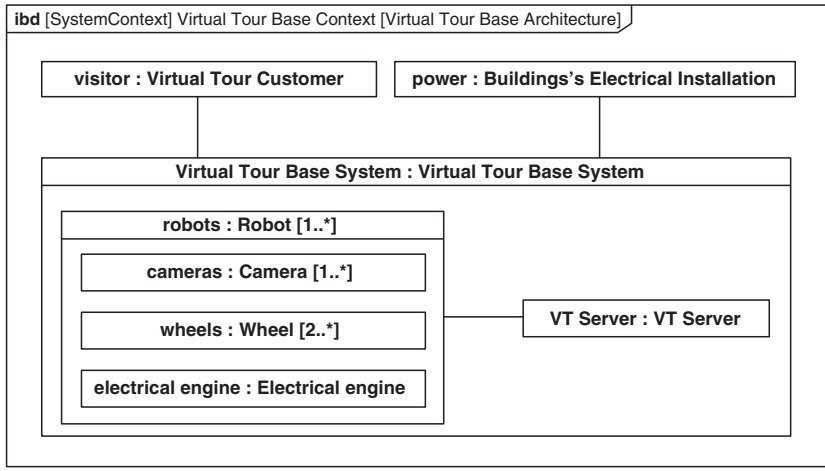

ibd [SystemContext] Virtual Tour Base Context [Virtual Tour Base Architecture]
visitor : Virtual Tour Customer
power : Buildings's Electrical Installation
Virtual Tour Base System : Virtual Tour Base System
robots : Robot [1..*]
cameras : Camera [1..*]
wheels : Wheel [2..*]
electrical engine : Electrical engine
VT Server : VT Server

Figure 9.10 Base architecture of the Virtual Tour system.

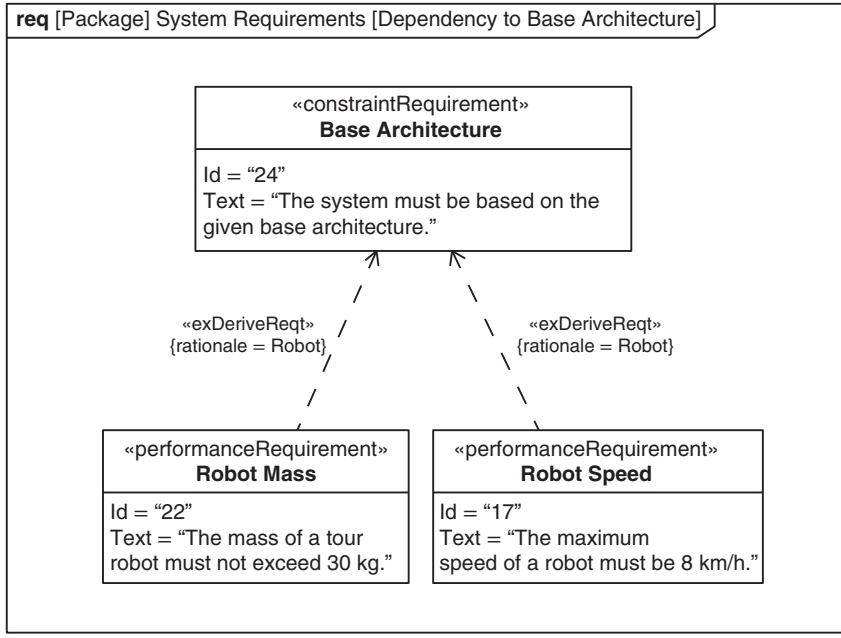

req [Package] System Requirements [Dependency to Base Architecture]
«constraintRequirement»
Base Architecture
Id = "24"
Text = "The system must be based on the given base architecture."
«exDeriveReqt»
{rationale = Robot}
«exDeriveReqt»
{rationale = Robot}
«performanceRequirement»
Robot Mass
Id = "22"
Text = "The mass of a tour robot must not exceed 30 kg."
«performanceRequirement»
Robot Speed
Id = "17"
Text = "The maximum speed of a robot must be 8 km/h."

Figure 9.11 Relationship of top level requirements to base architecture.

The base architecture constrains the solution space and makes implicit technical decisions explicit. Therefore, it is also a good place to spot the potential for disruptive innovations. You can put a question mark over the decisions in the base architecture.

In practice, a base architecture description is a set of block diagrams and additional textual descriptions. At a minimum, it is a context diagram, a product tree, and an architecture block diagram. Figure 9.10 depicts the context and architecture elements in a SysML internal block diagram. Figure 9.12 depicts the product tree. For each actor and part of the architecture, a textual brief description is provided in Table 9.1.

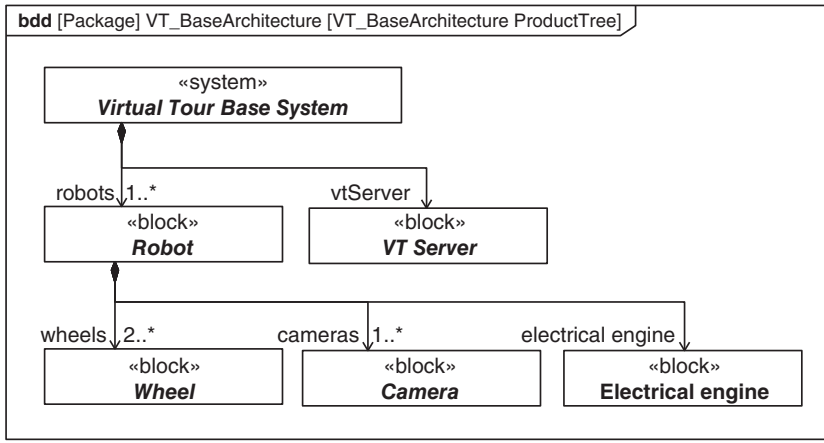


Figure 9.12 Product tree of the Virtual Tour base architecture.

Table 9.1 Brief description of base architecture elements of the Virtual Tour system

Property	Brief description
robots : Robot [1..*]	The robots are physical systems controlled by the tour server or the tour customer. The robots move on wheels and use cameras
cameras : Camera [1..*]	At least one camera to record the environment of a robot for the users and to navigate
wheels : Wheel [2..*]	At least two wheels to move the robot through the building
vtServer : VT Server	One server to control all robots of a single Virtual Tour system
electrical engine : Electrical engine	The robot is driven by an electrical engine

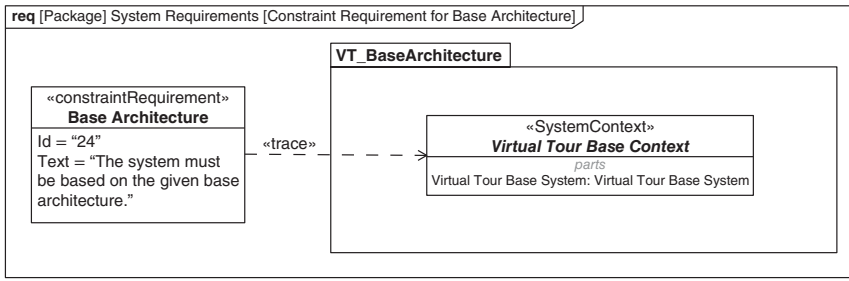


Figure 9.13 Virtual Tour system constraint requirement for base architecture.

You could also formalize the base architecture with constraint requirements. The base architecture provides technical constraints for the system under development. Depending on your requirements engineering process, you could consider to transform the constraints of the base architecture to constraint requirements in your requirements model. Otherwise, you must ensure that the base architecture is always part of your requirements model and documentation.

A pragmatic approach is a single constraint requirement with a trace relationship to the root element of the base architecture (Figure 9.13).

9.3 Cohesion and Coupling

Cohesion and Coupling is a common applied principle for systems. Figure 9.14 depicts both dimensions of the principle.

Cohesion Principle
A part of a system should be cohesive as possible, that means all elements inside should belong together.

Simply spoken, the cohesion principle says that things that do similar tasks should be at the same place. Short ways make them more effective, and changes have local effects. The coupling principle is related and acts contradictory. That means, if you improve the cohesion, you worsen the coupling and vice versa. The right application of these principles requires good skills of the system architect.

Coupling Principle
A part of a system should be loosely coupled as possible, that means it has a minimum number of weak dependencies to other parts of the system.

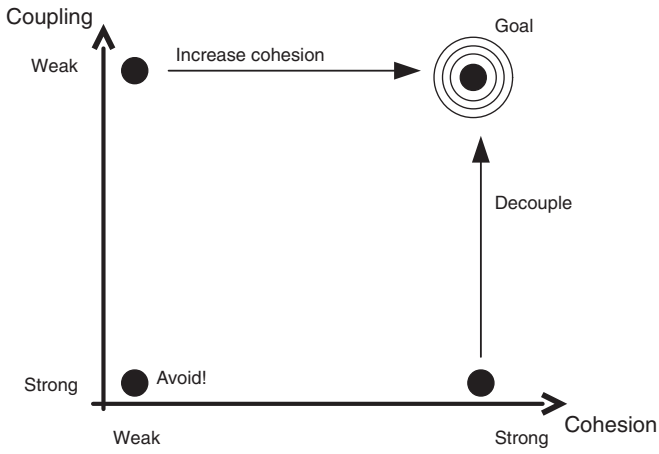


Figure 9.14 Cohesion and coupling. Copyright oose Innovative Informatik eG.

A dependency between parts could be explicit or implicit. Parts are explicitly coupled, for example, if they have a mechanical or an information technology (IT) data exchange connection. The coupling is stronger the more a part must be updated when the connected one is changed.

Parts are implicitly coupled when they have a logical dependency without having an explicit connection. For example, if a behavior implemented in a software depends on a physical parameter of a part. These hidden links are critical since they are hard to identify and could lead to undesired behavior and malfunction of the system when one part is being updated without the other one.

The notion of the coupling principle is two make the system more modular. Parts could be updated or replaced without having a (strong) impact on other parts.

The cohesion and coupling acts contradictory. If you increase cohesion, which is good, you also increase the coupling, which is bad. A strong cohesion leads to more parts, with each having only a small set of the overall provided functions. Fewer functions fit more to a strong cohesion criterion of a single part. In consequence, you have more parts to cover all functions. More parts of the system require more relationships, which means more coupling.

If you decrease coupling, you also decrease cohesion. The extreme case is a system that consists of a single assembly. The coupling on that level of detail is zero. But the cohesion is extremely weak, because that assembly provides all functions of the system even if they are not closely related and have nothing in common.

Note that the cohesion and coupling principle appears on different levels of the architecture. The single assembly in our example above contains parts and those parts could have a strong coupling and cohesion on their level of detail. And each

part of the parts could again include parts and so on. The cohesion and coupling are different on each level.

The origin of this principle is the software engineering discipline. Larry Constantine published the cohesion and coupling metrics in his article "Segmentation and Design Strategies for Modular Programming" in 1968 [49]. The principles are also known in non-software disciplines. Karl Ulrich covers the coupling of mechanical interfaces and related dependencies between the parts in the paper "The role of product architecture in the manufacturing firm" [253].

9.4 Separation of Definition, Usage, and Run-Time

The definition of elements and the definition of their usage in a specific context, and the definition of their structure and relationships at run-time are separate aspects and should be handled separately.

The definition of an element is a blueprint defining the structure and behavior of the element. A simplified example is a screw. You can define the head, shaft, material, length, and diameter of the screw. For common elements like screws, the definition could be published in a parts catalogue. It is the blueprint for all screws of that type and not a representation of a single concrete screw from the real world. In SysML, we specify the screw definition with blocks in a block definition diagram (Figure 9.15).

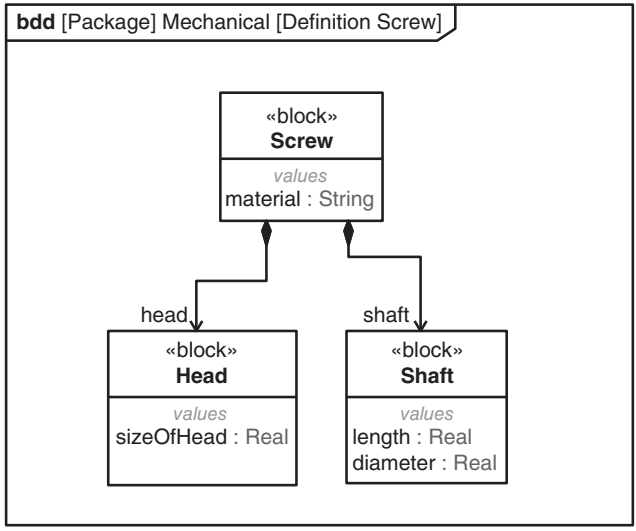


Figure 9.15 Definition of a screw with SysML.

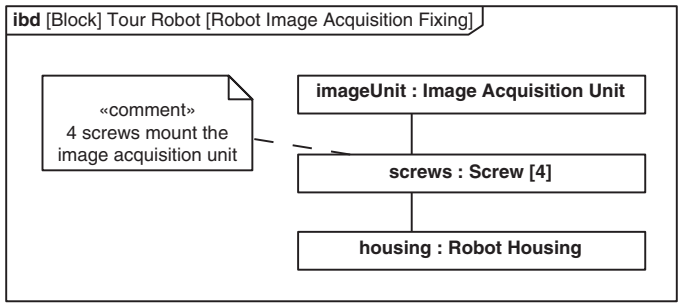

ibd [Block] Tour Robot [Robot Image Acquisition Fixing]
«comment»
4 screws mount the
image acquisition unit
imageUnit : Image Acquisition Unit
screws : Screw [4]
housing : Robot Housing

Figure 9.16 Usage of a screw with SysML.

The usage defines the application of a screw in a specific context. For example, that our screw is used to mount the image acquisition unit on the housing of the tour robot. The definition of the screw constrains the usage level. You can only use the elements according to their definition. In the other direction, the usage level has no impact on the definition. In SysML, we define the usage within a context with the internal block diagram (Figure 9.16).

The run-time level specifies the links and properties of elements at a specific point of time during the run-time of the system. Figure 9.17 shows a concrete mounting of an image acquisition unit at a tour robot at a specific point in time. Robot *xsnM871* has one missing screw.

Be aware of the different levels. It is easy to mix them up. For example, the usage level could also be part of the definition of an element. Figure 9.18 shows the internal block diagram of the screw. It specifies the connection of the head and the shaft. It is a usage of the blocks *Head* and *Shaft*, but it is part of the definition of the block *Screw*.

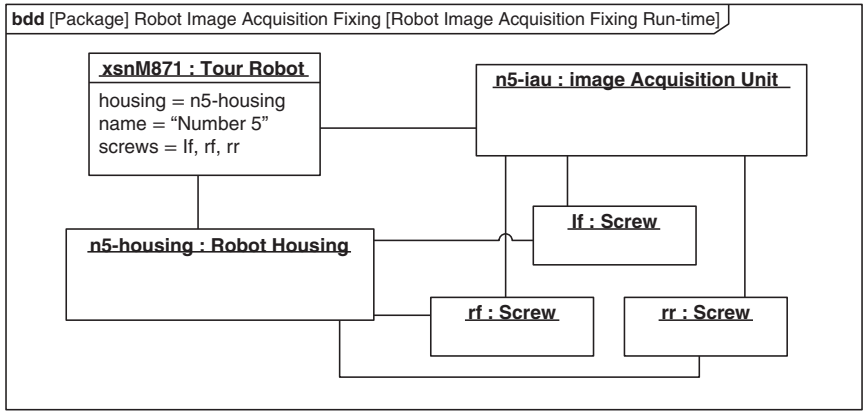

bdd [Package] Robot Image Acquisition Fixing [Robot Image Acquisition Fixing Run-time]
xsnM871 : Tour Robot
housing = n5-housing
name = "Number 5"
screws = lf, rf, rr
n5-iau : image Acquisition Unit
lf : Screw
n5-housing : Robot Housing
rf : Screw
rr : Screw

Figure 9.17 Run-time setting of the screw with SysML.

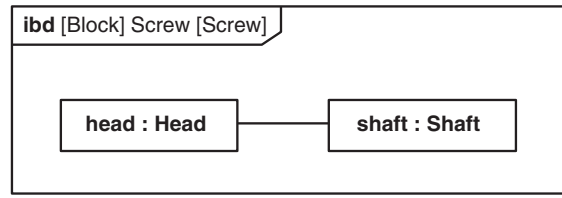


Figure 9.18 Internal structure of the screw.

SysML follows the principle of the separation of definition, usage, and run-time aspects. The block definition diagram defines the blocks. The internal block diagram specifies the usage in a defined context. Activities define behavior, call behavior actions the usage of activities, and so on. It is important to know the three levels, to handle them separately, and to know the relationship between the levels.

9.5 Separate Stable from Unstable Parts

Any model, documentation, or real system has stable and unstable parts. The stability property refers to the number of changes of the entity. A stable part is seldom changed, and an unstable part is often changed. It is a best practice to separate stable parts from unstable parts, and stable parts should not depend on unstable parts, because they adopt the unstability.

In the system architecture model, you have, typically, stable technical principles and concepts and unstable concrete physical part definitions. If you follow the principle to separate stable from unstable parts, you get an logical architecture covering the stable technical concepts and principles, and a product architecture covering concrete physical blocks specifying the system-of-interest. Another example is the separation of concerns using layers. See Section 11.5 about layered architectures.

9.6 The Ideal System

The ideal system is a tool to narrow the solution space to find a system that really satisfies the user's demands. The ideal system fulfills the requirements of the user without even existing. That sounds ridiculous, but this line of thought is helpful. Reflections on this obviously unattainable goal restrict the search space for a solution and direct the focus of the system development in a user-friendly direction.

For this purpose, a concrete example taken from [267]: Consider the locking system of a car. Let's start at the time when it was necessary to put a key into the lock and to turn it to unlock a door of the car. Is this that what you want as a user? With all the lovely side effects that the lock is frozen in winter or the paint of the

car is being scratched from the key. As a user, you don't want all these. You want to protect your car against theft and misuse. The closing system definitely existed and was very present.

Today's cars have, typically, a central locking system, which can be opened with a remote control on the key. One click locks or unlocks the doors. The locking system is less existing for the user and provides the same functionality as the key-based system. It is still not an ideal system and has some inconveniences. For example, you notice after fasten the seat belt that the key is still in your pocket and you need it to start the car.

The latest technology is already is very close to the ideal system. Once you use the door handle of the car the car recognizes whether a RFID key is nearby and opens the car when it is detected. With a fingerprint scan on the shift lever, the motor can be started. For the user, the system is barely existent, but the desired functionality is available.

Consider the systems that surround you and think about how these systems evolved over the years. Mostly they strive toward the ideal system. The principle is valid for most systems. The user requests functionality and not the system itself. The guiding principle is as follows: Focus on solutions that are as inexistent for the user as possible.

The ideal system is a principle from the TRIZ methodology [17]. TRIZ is an acronym that stands for теория решения изобретательских задач (триз), which is Russian and means "Theory of Inventive Problem Solving" (TIPS). This theory is based on a systematic methodology for innovative processes. The father of TRIZ – Genrich Soulovich Altshuller (*15.10.1926–†24.09.1998) – was convinced that inventions are no coincidence. He has analyzed thousands of patents and derived TRIZ from his insights. The ideal system is one pattern he has observed.

9.7 View and Model

An important principle in modeling is the separation of view and model. The model is the source of the information. The modeling language defines the data structure and the semantic of the model elements. The data structure is also called abstract syntax. The left side in Figure 9.19 shows the abstract syntax of the SysML use case and include relationship. The abstract syntax is not a notation but a definition of the data structure. The storage format could be XMI, a XML language for SysML and UML models [187].

The view is a textual or graphical representation of the model. Typically, a modeling language provides a notation for the model elements – the so-called concrete

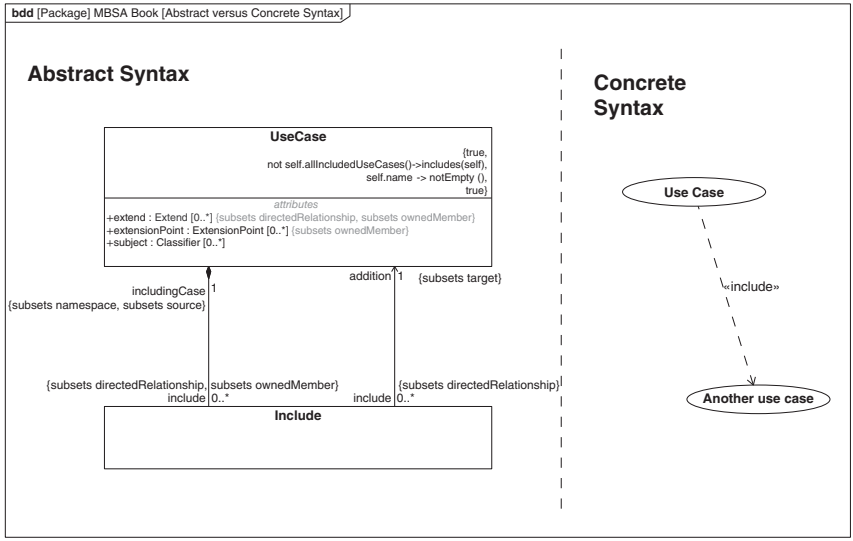


Figure 9.19 Abstract syntax of a use case.

syntax. The right side in Figure 9.19 depicts the concrete syntax of the SysML use case and include relationship. However, not every modeling language provides a graphical notation. For example, the "Business Motivation Model" (BMM) [186] – a standard of the OMG for modeling vision, mission, strategies, goals, and more for businesses – does not provide a notation, but only the abstract syntax and semantics.

The view should not contain any additional semantic that is not part of the model. That means you can remove all views without losing relevant information. See Section A.2 about the separation of view and model in SysML.

If your primary focus of modeling is the communication between the development team members, the view is more important than the model. You can even create views without a real model like flipchart sketches or drawings in slideshow applications. That is not decent modeling, but more painting. If the specification, analysis, or simulation is your primary focus, then the model is more important than the graphical views. You could even discard the graphical notation and work only with the model.

Naturally, people look on the view artifacts and not the model. In a modeling environment like SysML, the view is also the editor to create or change the model elements. The "real" model could easily get out of the scope. Be aware of it and handle the view and the model according to your needs.

9.8 Diagram Layout

In system modeling, the graphical representation of the model data is an important part. It definitely makes a difference how you present your data. The human brain looks for graphical patterns and not for all the little details. That makes life much easier. For example, you can change the letters in a text while keeping the pattern structure, and you could still read the text:

Mdoel Baesd Setmsys Ennneigierg mekas my procjets mroe ecffetive.

Similar things happen when you look at model diagrams. You will see graphical patterns. Unfortunately, such patterns are not well known and described. Here is a huge potential to be more effective. For example, it makes a difference if you layout a product tree in a block definition diagram top-down, bottom-up, or in a network layout style (Figure 9.20). We prefer the top-down layout. It emphasizes the breakdown of the product (product breakdown structure [PBS]).

Some notes about diagram layouts:

- In the authors' culture, the reading direction is from top to down, from left to the right. It is different in other cultures. Layout your diagram conforms to the reading direction of your stakeholders.
- A diagram should have a printable layout (A4 or letter format).

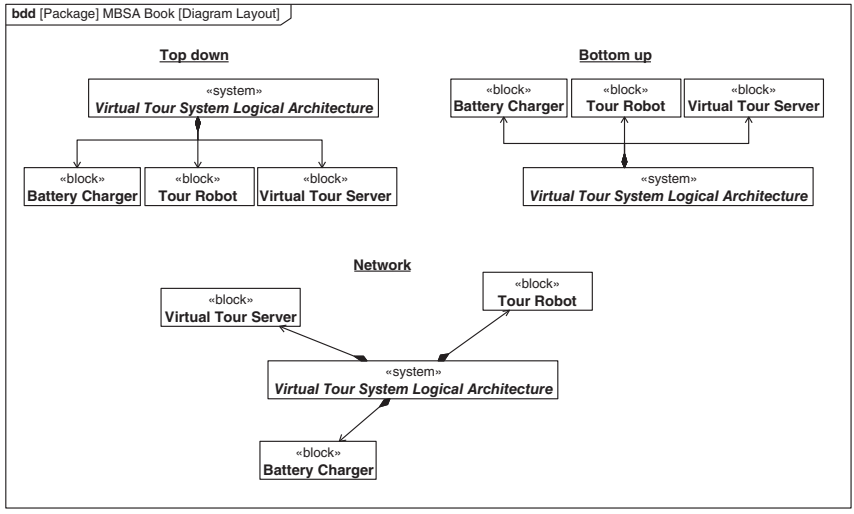


Figure 9.20 Different layout directions.

- Rarely use colors in your diagrams. Some people are color-blinded, and monitors, printers, and beamers present colors differently. Instead, use gray colors to emphasize specific aspects if you need different colors in your diagram.
- One diagram has one purpose.
- Look through the eyes of the model reader when you create a diagram and not through the eyes of a model builder.
- Clearly separate diagram creation from model building. See also Section 9.7.

9.9 System Model Structure

We will look at system model structures from the viewpoint of a SysML model. However, the basic concepts are valid for all system model kinds.

At first glance, it seems to be simple to define the package structure of a SysML model. However, you will often get problems with an implicit built structure. A model has many – orthogonal – aspects and abstraction levels that could be mapped into the package structure, for example, domain, modeling, or organizational aspects. You can easily mix up the orthogonal aspects. Probably you know that problem from organizing your computer's hard disk with a sophisticated directory structure.

The "MBSE Challenge Team SE^2 for Telescope Modeling" describes a best practice for the package structure in the "MBSE Cookbook" [125]. Figure 9.21 shows the top level of the system model package structure.

The root package represents the complete system model. If you model variants (see Section 18), the next level has three packages: one for the configurations, one for the core, and one for the variations. If you do not model variants of your system, you skip the package structure of this level and directly proceed with the structure that is used in the core package. See Section 18 about how to model variants and how to manage the configuration, core, and variation packages.

On the next level in the core package, we separate the different modeling aspects like system context, requirements, and logical architecture. The list in the figure is not complete. You will have your own appropriate aspects for your projects. The prefix of each package refers to the enclosing namespace. Typically, you have many packages of the same name in the model, for instance "Requirements." The prefix shows the context of the specific package.

The logical architecture package contains the structural elements of the systems, that means, the physical blocks. Each block that has a detailed description has its own package on the next level, for example, the subsystem "TourBot." You treat the package like the system root package and create the same package structure inside. The package "TourBot_Requirements" in Figure 9.21 contains

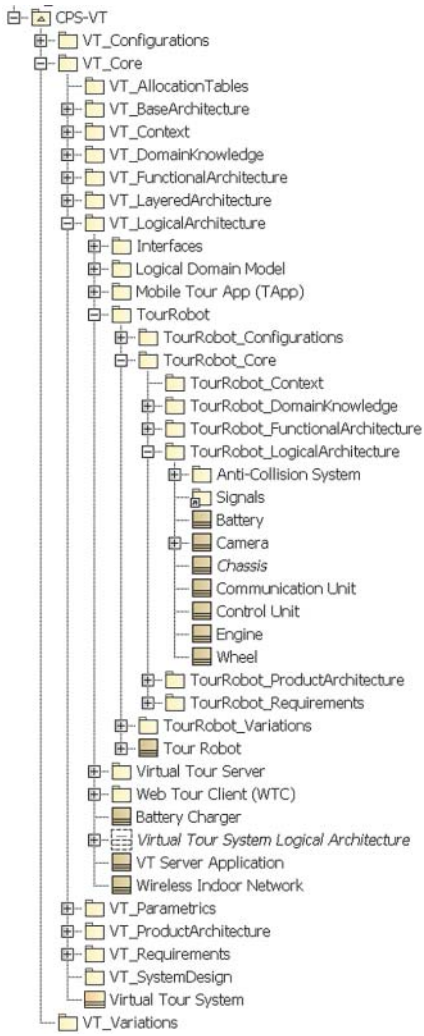


Figure 9.21 Top level system model structure.

all requirements that directly relate to the subsystem "TourBot." Again, there is a logical architecture package that contains further packages with the same structure. If you strictly separate all TourBot specific elements, the TourBot package is a complete (sub-)model of its own for the tour robot, including requirements and the architectures.

The package structure is straightforward. It works for models of any size and gives the model builder and users a good orientation. The structure could be used for models of any size. The model of a telescope system from the MBSE Challenge Team SE^2 is a good example of the application of this concept [125].

9.10 System Architecture Principles

Often principles come across as very simple and obvious, and you wonder what is so special about them. In that case, the principle is already second nature to you, and you will probably follow it automatically or deliberately violate it when it is necessary and helpful.

Edward Crawley et al. list 26 principles for system architectures in their book [50]. We will not repeat them all here, but refer to the book. We will pick out two related principles as examples.

The "Principle of Decomposition" covers two aspects. First of all, it is very normal to do a functional and structural decomposition in systems engineering. That it is so natural carries a risk, and that concerns the second aspect. There are various ways, for example, in which a system is decomposed structurally. This should not just happen but is an explicit decision of the system architect. The decomposition leads to internal boundaries and sets the borders where to define interfaces. It could have an impact on the organization, on the configuration management, on the quality of the system, and so on.

The "Principle '2 Down, 1 Up'" is related to the "Principle of Decomposition." It states that the goodness of decomposition cannot be assessed if you do not know the decomposition of the next level. That level shows how to best decompose the entities on level 1 to follow the cohesion and coupling principle (Section 9.3), which means maximizing the internal couplings (cohesion) and minimizing the external couplings.

Following the principle means that starting from level 0, you create a decomposition level 1 and then another level 2. The level 2 results in findings for level 1, which is then adjusted accordingly.

If there is no need to keep level 2, it can only be considered as a tool to create level 1 and then discarded. Figure 9.22 depicts the principle but without showing the relationships between the entities which are needed for the evaluation.

9.11 Heuristics

9.11.1 Heuristics as a Tool for the System Architect

Heuristics are an important tool for the system architect. This has been pointed out very thoroughly by Maier and Rechtin [164] who explain that heuristics are condensed experiences, phrased in an easy way that allows communicating them to others. In that sense, we would like to contribute our own set of experiences phrased into heuristics. But we will not even try to to cover the aspect of heuristics in the same proficiency as Maier and Rechtin, because this attempt would most certainly fail.

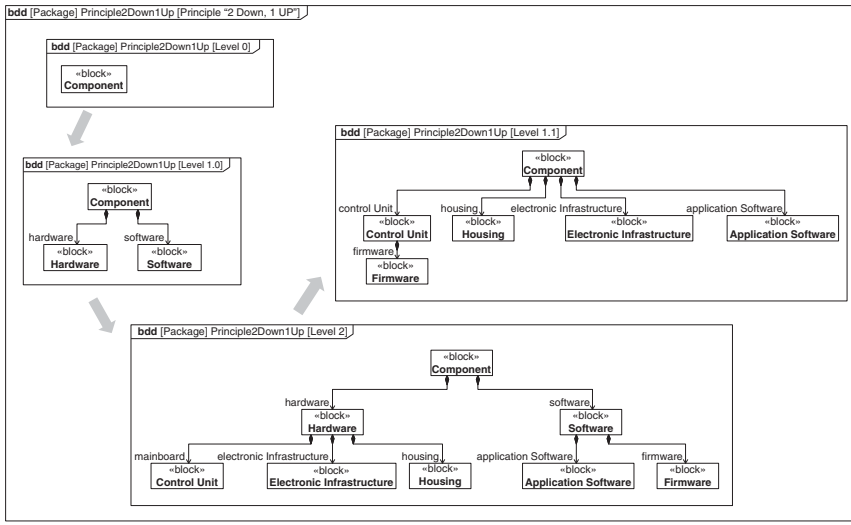


Figure 9.22 Example Principle 2 Down, 1 Up.

Heuristics are similar to principles and patterns. A heuristic is based on experience and is a general rule of thumb for something that works. A pattern is a solution to a specific problem, and a principle is a guideline on how to do something. The three terms are not sharply separated, which is not problematic.

We highly encourage system architects to make use of the powerful tool that heuristics are, in contrast to pure equations and other deductive findings that may solve well-defined problem statements but could easily fail in addressing the multi-domain challenges encountered in systems architecting. Collect your own experiences and condense them into heuristics. Ensure that the heuristics are available when needed. To all those who like to know more, we highly recommend the book "The Art of Systems Architecting" by Maier and Rechtin [164]. For those who are familiar with heuristics in general, we now provide our very own heuristics:

- **Make it lovely, not weird:** Architecture descriptions should be nice to work with and should look good. Sometimes it is just the considerations about diagram layout from further above that will make this happen. Sometimes it is also the use of illustrative pictures or the review by a colleague with a good eye for aesthetics that will make this happen. Our experiences with both succeeding to convince people with nice architecture descriptions and failing with too user-unfriendly ones tell us that the effort spent in making communication materials look nice will often pay back.
- **Abstraction cuts the Gordian Knot:** Master complexity with abstraction. If the system gets larger or more complex, then the level of abstraction of the top

level architectural views has to be increased. There is only a limited amount of complexity an architecture core team can deal with. A possible way out is to choose an abstraction level as high as the system's complexity can still be governed on that level.

- **Choose the Upper deck:** If in doubt whether to model on a higher abstraction level or a slightly lower one, always choose the higher abstraction level (this heuristic is also known as "Jesko's law").
- **The Modeler should stick to ending fast:** Never model so deep that the content of system architecture is redundant with content of engineering documentation of the engineering disciplines. The cobbler should stick to his last, and the architect should stick to the system-level.
- **Go where the Money is:** Make architectural developments and refactoring of existing system architecture in projects with enough resources and not outside any market-driven activity. The projects whose business case predicts high profit also have the resources to realize architectural improvements. By contrast, an architect sitting alone in the corner of a department may find genius architectural improvements but may lack the lobby for getting them implemented.

9.11.2 Simplify, Simplify, Simplify: Strength and Pitfall

At the end, let us come back to Maier and Rechtin [164] once more. One very central heuristic they mention is "**Simplify. Simplify. Simplify.**" (which is very close to the commonly encountered "keep it simple"). This heuristic applies in many contexts and should often be the driver of the system architect's action. If there is a choice between a simple concept or a complex concept, then the simple one is most of the time the better one. If a simple sketch of the problem is sufficient to solve the problem, it is not necessary to make a more detailed or more complex sketch about the problem. For example: If an envisioned new product is known to have 50% more interfaces to external systems than its predecessor, then it is probably not necessary to make the detailed interface specifications to find out that the system architecture workload for the given product will be higher than the one of its predecessor. The system architect in charge of initiating a rough effort estimate should thus maybe start making a list of all interfaces rather than diving into one or two of them.

However, the "Simplify. Simplify. Simplify." comes with one pitfall: If a system is complex, then it should be handled by simple means, but not by too simple means. Systems become complex when they have to satisfy complex requirements. The complexity has to be controlled somewhere. The strength of model-based systems architecting is the creation of different views on the same complex system – each of them showing only one aspect and thus hiding the complexity for the stakeholders using this view. The model, however, corresponds to the complexity of the real

system to a degree that is sufficient close for solving the problems that arise from the complexity.

Models of complex systems will probably become complex as well. While the system architects have to govern the complexity, the stakeholders have to focus on the aspects of the system that are relevant to them via the appropriate views. They have to trust the system architects that they will control the overall system and have to be able to live with their limited view on the system. Critics claiming that the system architects are creating too much complexity are often not following this principle of trust. They like to understand the full model themselves, which is often no longer possible with complex systems.

Often mentioned quotes carry a lot of truth: "Everything should be made as simple as possible, but not simpler." (Albert Einstein) [46].

10

Model-Based Requirements Engineering and Use Case Analysis

The requirements engineering and use case analysis is not part of the system architect's activities. However, the architect has close interfaces to the requirements engineering discipline, for example, when working with functional architectures (Chapter 17). This chapter gives a brief description of requirements engineering and use case analysis. For a more detailed description, we recommend a common book about requirements engineering, for example, Robertson and Robertson [209].

In Section 10.1, we define the most important requirements and use case terms. Then, we present in Section 10.2 the requirements and use case analysis methodology steps of the "Systems Modeling Toolbox" (SYSMOD) [267]. The SYSMOD methodology defines common methods and is not specific to any modeling tool. Finally, we take a look at two related methods. Section 10.3 covers the Storyboard Activity Modeling for Systems (SAMS) method that provides an illustrative approach to identify use case activities with stakeholders. Section 10.4 presents the Use Case 2.0 approach that works well with agile frameworks (see also Chapter 16).

10.1 Requirement and Use Case Definitions

Figure 10.1 depicts the requirement terms, and Figure 10.2 the use case terms we use in this book. Of course, the requirements engineering and use case analysis domain have much more terms, but they are out-of-scope of this book.

The source of the requirements is the list of stakeholders. A stakeholder has an interest in the system due to its impact to one or more processes operated by the stakeholder. The needs of the stakeholders are expressed by their concerns. A concern denotes an interest of any kind into the system-of-interest and elicits to a set of requirements. A special concern is the purpose specifying what a stakeholder intends to achieve using a system.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

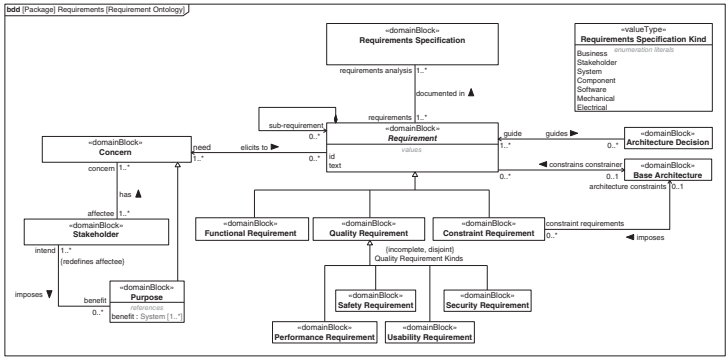


Figure 10.1 Requirement ontology.

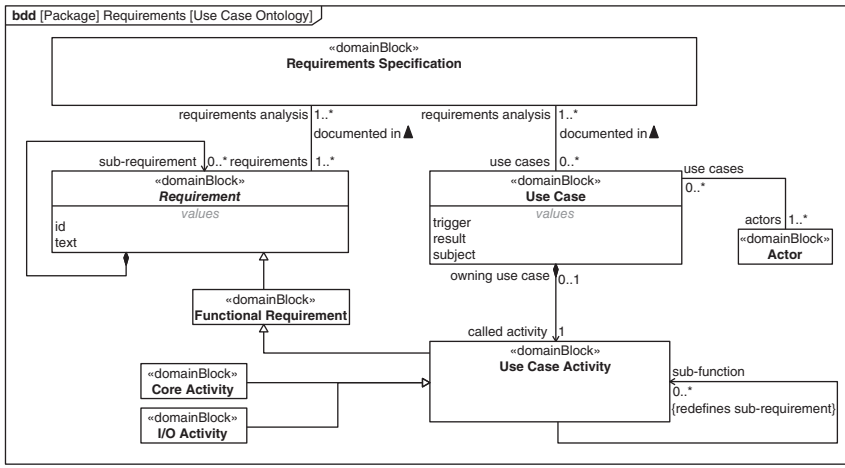


Figure 10.2 Use case ontology.

A requirements specification is a collection of requirements. There are several kinds of requirements specifications like business requirements specification, stakeholder requirements specification, system requirements specification, and so forth. The different kinds listed in the RequirementsSpecificationKind enumeration entity are not intended to be complete, and the names are used in projects differently. A manifestation of a requirements specification can be a document, but also a SysML requirements model or a set of requirements in a requirements management tool.

A requirement specifies a capability or condition that must or should be satisfied by a system. The definition is similar to the definition of a requirement in SysML [189]. A more detailed definition that does not conflict with this definition can be found in the INCOSE Handbook [265]. There are many more definitions, most of which are not conflicting but just emphasize different aspects. We limit ourselves to a simple definition since requirements are only a marginal topic in this book.

The requirements can be categorized into different requirement kinds. There are quite a few different categorizations in the literature and standards. We differentiate between functional, quality, and constraint requirements. The quality and constraint requirement categories are also called nonfunctional requirements.

- A functional requirement specifies functionality that the system shall provide.
- A quality requirement specifies how well the system shall perform during its use. The quality requirements are, in particular, important for the architecture. For example, it typically requires different architectures if you would like to build something that moves with 50 or 500 km/h or a system that handles 1000 or 10 000 000 users at the same time. Figure 10.1 shows four subcategories for

quality requirements: safety, security, performance, and usability requirements. A more comprehensive list also with subcategories is provided by ISO/IEC 25010:2011 [113].

- A constraint requirement specifies a condition not needing to be implemented, but must be complied with. For example, a physical law, a deadline, a budget, or architecture or technical decisions that are already fixed at the beginning of the project.

Use cases are closely related to requirements. Their ontology and relationship to requirements are depicted in Figure 10.2.

Use cases are documented in requirements specifications like the requirements. A use case specifies a coherent interaction of actors with a subject. The interaction is initiated by a trigger from the outside of the subject and ends with a result that is of value for the actor or other stakeholders of the subject. An actor represents a role outside the subject and can be a user or another system. The use cases are the functional interfaces between the actors and the subject. They are just the wrapper of a behavior defined by a use case activity, which defines a requested function and can be further decomposed by other called use case activities (functional decomposition). Typically, the subject of a use case is the whole system. We use it differently for the specification of an aggregated system in Section 17.12.

A use case activity is also a functional requirement. Through functional decomposition, it can be defined small enough to fit the respective rules of a functional requirement.

It is methodologically valuable to distinguish use case activities into two categories. The I/O activity defines a functionality of the system that implements the exchange with an actor, for example, the input of data, but also mechanical functions, such as the output of energy or heat. The system activity defines a functionality that does not directly involve an actor. I/O functionality is, typically, more dependent on technology than system functionality which depends more on the domain. The separation of these kinds is an application of the separation of stable from unstable parts principle (Section 9.5). It is also used by the Functional Architecture for Systems (FAS) method (Chapter 17).

10.2 Model-Based Requirements and Use Case Analysis from the MBSA Viewpoint

Hereinafter, we briefly describe each task with a focus on the interface to the system architecture discipline. The interface between requirements engineering and system architecture is often underestimated in system development. Typically, in focus is the derivation of the architecture from requirements. That

requires a close communication between architects and requirements engineers to resolve unclear and conflicting requirements. Additionally, like shown in the zigzag pattern (Section 9.1) technical decisions of the architect lead to new requirements. Altogether, requirements engineers and system architects must closely collaborate and should not just communicate via documents and models.

Since requirements and architecture are closely related and artifacts are linked, we recommend having the linked artifacts in the same physical model (see also Chapter 6). Elements of the architecture are linked to requirements they satisfy, and requirements are linked to elements from their base architecture that have led to those requirements.

10.2.1 Identify and Define Requirements

The identification and description of requirements is a wide topic. It is about the elicitation, structuring, documentation, wording, the management of requirements, and so on. We will not cover that here and assume that you have your own processes to handle your requirements in your projects. Other books cover requirements engineering in detail, for instance [209].

In our book, we use SysML as the modeling language to specify the system model and, in particular, the system architecture. SysML supports the modeling of requirements (Section A.5). There are three noteworthy scenarios for the linking of requirements and the system architecture in a SysML environment:

1. The requirements are modeled with SysML and part of the same physical model as the architecture. The elements are linked using SysML elements like the satisfy relationship. There is a traceability path from the architecture to the requirements.
2. The requirements are modeled outside the physical SysML model of the architecture, for example, stored in a separate repository managed by a requirements management tool. You have an adapter to transfer information about the requirements from their original model to the system model in SysML. The requirements in SysML are placeholders for the original requirements in the external physical model. They have SysML relationships to elements of the system architecture. There is a traceability path from the requirements to the architecture.
3. The requirements are stored outside of the physical SysML model in another model or document. The relationship between architecture elements and requirements is loosely documented in the documentation of the architecture. For example, a matrix in a textual spreadsheet could be used to manage the relationships. Although there is a traceability path from the architecture to the requirements, the path could not be automatically analyzed since it is not established by model elements.

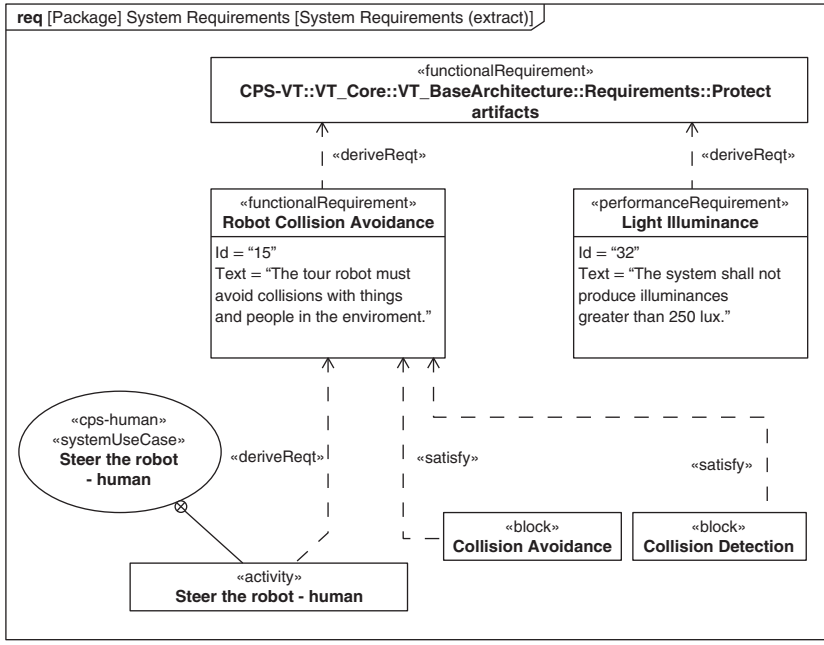


Figure 10.3 Requirements in a SysML model.

We prefer scenarios 1 and 2, because the traceability path is completely established by the model.

Figure 10.3 shows a set of requirements in a SysML diagram, including some relationships. The deriveReqt relationships define that a requirement is derived from another requirement. They are mainly used between the levels of the zigzag pattern described in Section 9.1. The satisfy relationship links the architecture with the requirements specifying that the source element satisfies fully or partly the target requirement.

See Section A.5 for more details about requirements modeling with SysML.

10.2.2 Specify the System Context

At the beginning of a system development project, it is necessary to identify the system boundary, external interfaces of the system-of-interest and interacting systems or humans, for example, Walden et al. [265]. A system context diagram in SysML can be used to describe these elements [267, 271]. Figure 10.4 shows a SysML internal block diagram (Section A.3.2) as a system context diagram, where the black line around the system-of-interest "Logical Virtual Tour System" depicts the system boundary, and the solid lines between the system and external

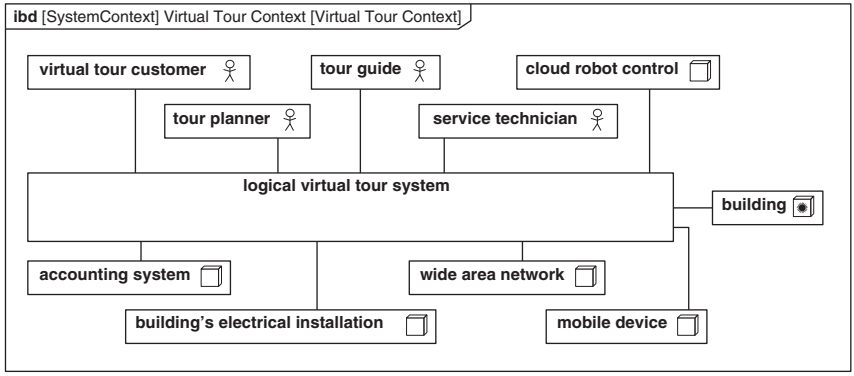


Figure 10.4 Virtual tour system context.

entities – the so-called system actors – depict interactions between the system and its context elements. All further analysis and architecting steps are assumed to describe the system-of-interest within this identified system boundary.

The actors are classified in different categories: human actors (sticky man) and nonhuman actors (boxes). Figure 10.4 shows also a special kind of a nonhuman actor. The box with the sun represents an environmental effect. Here, it is the condition inside the building that interacts with our system-of-interest (temperature, humidity, light, …).

Figure 10.4 depicts only some elements of the system context. The system context includes more details like system interfaces, actor interfaces, and internal structures of the system-of-interest and the actors. Typically, it is a task of the system architect to detail the system context in close collaboration with the requirements engineer. However, once the information is in the model, you can create simple and extended system context views of the same model information. While the simple system context is more suitable for nontechnical stakeholders, the extended system context is for technical-oriented stakeholders like the engineers of the system-of-interest. Figure 10.5 depicts an extended system context view in a SysML internal block diagram.

A comprehensive system model has multiple context specifications. For example, the operational context depicts the context entities of the operational phase or the verification context as described in Section 11.8.3. The general system context summarizes all context information.

10.2.3 Identify Use Cases

Use cases are important artifacts in systems development. They direct the focus on user perspectives of the system. The outside-in view supports the development of

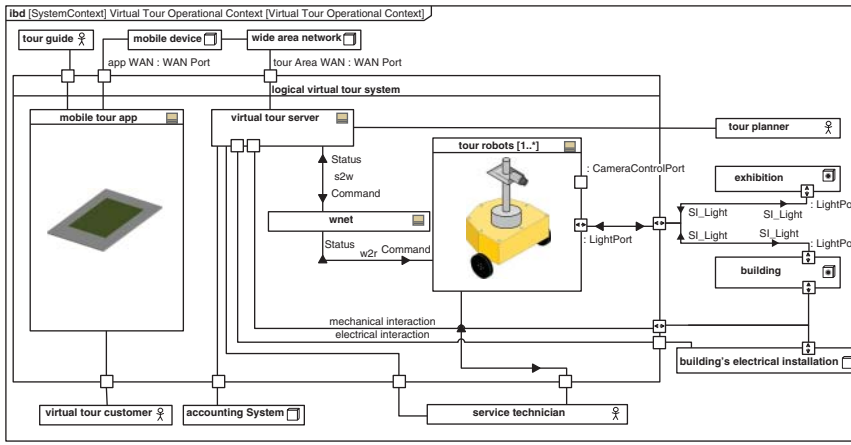


Figure 10.5 Extended system context.

systems that really satisfy the needs of the users and stakeholders in contrast to the widespread engineering inside-out view of the system. The use case perspective is also well understandable for people having no engineering background.

Use cases are a wrapper around the system's functions, defining the preconditions and postconditions as well as the trigger of the use case occurring at the system boundary, and the result returned to entities of the system context (typically, the actor who triggers the use case). Use cases have been extensively discussed in the literature of model-based software and systems engineering [128, 267]. Textual descriptions of system operations [25] or scenarios, for example, Pohl [201], can also be considered as use case descriptions. Storyboards or story-telling approaches can be used to elaborate use cases (for example, Section 10.3 or [161]).

Use cases are phrased from the actor's perspective. For example, the use case "Book a tour" is phrased from the perspective of the virtual tour client (Figure 10.9): The virtual tour client wants to book a tour. The reading pattern is: The ⟨system actor⟩ wants to ⟨use case name⟩.

An official definition of a use case is given by the Unified Modeling Language (UML) specification [188]: "When a UseCase applies to a subject, it specifies a set of behaviors performed by that subject, which yields an observable result that is of value for actors or other stakeholders of the subject." That definition defines the model element "UseCase" and does not contain any methodology aspects. They are added, for example, by the SYSMOD methodology [267, 269]: The system use case always has at least one actor, is started by one trigger coming from the system context, and ends with a result that fits the intention of the trigger. The behavior between trigger and result is temporally contiguous, which means there is no temporal interruption provided by the system (temporal cohesion).

The most important rule for a system use case is the temporal cohesion. The system use case is a specification of a complete collaboration between actors and the system. For example, the use case "Book a tour" ends with a booked tour. That is why the actor "Virtual Tour Client" has triggered the use case. The result satisfies the actor and ends the interaction. A function like "Select available tour" is not a use case. An actor won't use the system just to select an available tour and stops the interaction when the tour is selected. That function is only a part of a more comprehensive use case.

A system actor that triggers or participates in a use case does not need to be a human actor. It is a little bit old-fashioned that use cases are mostly described in combination with human actors. Nowadays, systems often communicate more with other systems than with humans. Figure 10.7 shows a use case that is only connected with an external system. We have observed in real projects that new use cases and requirements were identified when applying the use case analysis with nonhuman actors.

A special use case kind is the continuous use case. It describes a use case that continuously delivers results. The trigger could be an external or an internal event, for example, a state switch. The continuous use case "Charge robot battery" in Figure 10.7 starts when the tour robot is connected with the charger station and stops when it is disconnected. The charger is not an actor of the use case, since it is part of the system, but the building's electrical installation which provides the power.

Use cases are identified by analyzing each actor of the system context. Most actors trigger or are involved in a use case. Although use case diagrams like Figures 10.6 and 10.7 are very common, we recommend only to use them if you need the graphical representation, for example, for workshops or documents. Use

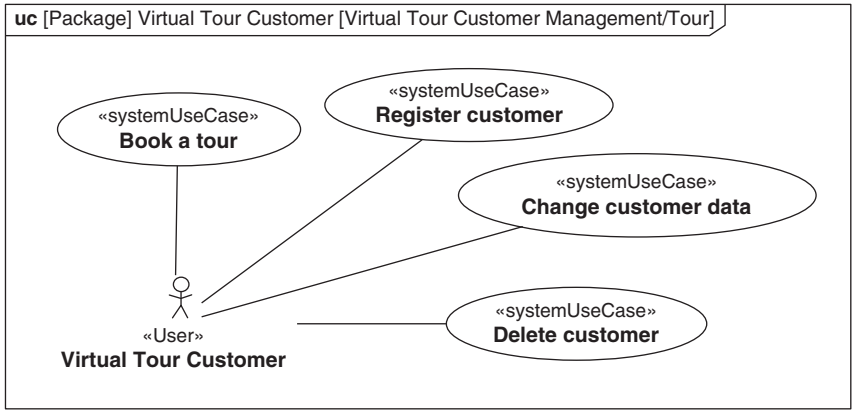


Figure 10.6 System use cases.

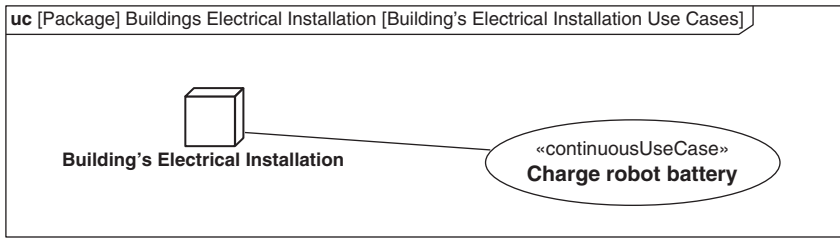

uc [Package] Buildings Electrical Installation [Building's Electrical Installation Use Cases]
Building's Electrical Installation
«continuousUseCase»
Charge robot battery

Figure 10.7 Continuous use case "Charge robot battery."

#	Name	Actors
1	Book a tour	Virtual Tour Customer
2	Change customer data	Virtual Tour Customer
3	Charge robot battery	Building's Electrical Installation
4	Delete a tour	Tour Planner
5	Delete customer	Virtual Tour Customer
6	Do a group tour	Tour Guide Virtual Tour Customer
7	Do a personal tour	Tour Guide Virtual Tour Customer
8	Edit a tour	Tour Planner
9	Define tour area geometry	Tour Planner
10	Plan a tour	Tour Planner
11	Register customer	Virtual Tour Customer
12	Register tour guide	Tour Planner
13	Add robot to VT	Service Technician
14	Change robot data	Service Technician
15	Configure robot	Service Technician
16	Lock robot	Service Technician
17	Steer the robot	
18	Steer the robot - human	Virtual Tour Customer
19	Steer the robot - machine	Cloud Robot Control
20	Unlock robot	Service Technician

Figure 10.8 Table view for use cases.

cases can be managed in tables more effectively than in a multitude of use case diagrams. A table is just another view of the model information (Figure 10.8).

A use case is a structural element that describes only the case, that is, name, actor, trigger, result, and so on. The actual behavior of the use case describes a use case activity. It describes the functions of the system that are executed to perform the use case.

10.2.4 Describe Use Case Flows

A use case is refined by a use case activity. This activity again contains calls to other use case activities that define the necessary functions of the use case. The calls of use case activities are also called use case steps. We phrase use case activities from the system's perspective, for example, "Confirm tour" instead of "Get tour confirmation." Note that the use case itself is phrased from the actor's perspective as well as the use case activity that represents the whole use case. Figure 10.9 depicts the use case activity tree of the use case "Book a tour."

The control flow specifies the call order of the use case activities. The object flow specifies the relationship of output objects to input objects of the use case steps. Figure 10.10 shows the control and object flows of the use case "Charge robot battery" in a SysML activity diagram. The control flow edges are dashed, and the object flow edges are solid lines. See Section A.4.2 for a description of SysML activity diagrams.

Each use case activity could be further refined by calls of use case activities. The refining activity – depicted by another activity diagram – is called by the use case step, which is a so-called call behavior action that could be recognized by the fork symbol in the lower right corner of the use case step rectangle.

We recommend modeling each use case step with a call behavior action and appropriate activity. The called activity could be empty and just have a name and short description if no further refinement is necessary. That way, it is easy to do a refinement later and to use activity trees as another view on the use case activities. Activity trees are a valuable view of the system behavior and a supporting tool to create a functional architecture (Chapter 17).

It is a good practice to separate behavior that handles system inputs and outputs from the "real" system behavior. The input and output behavior is more unstable

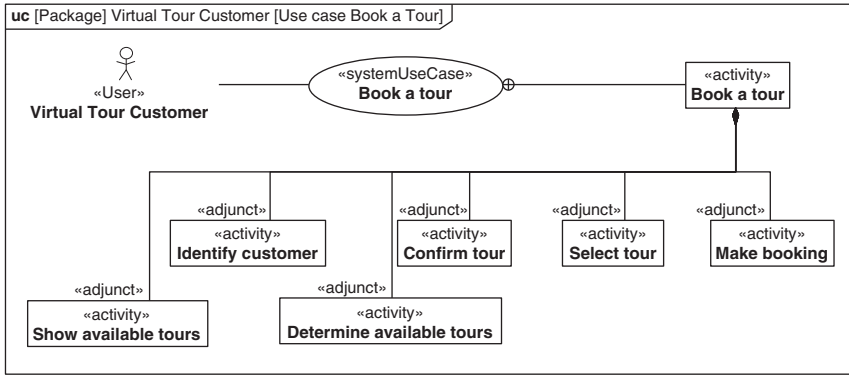


Figure 10.9 Use case activity "Book a tour."

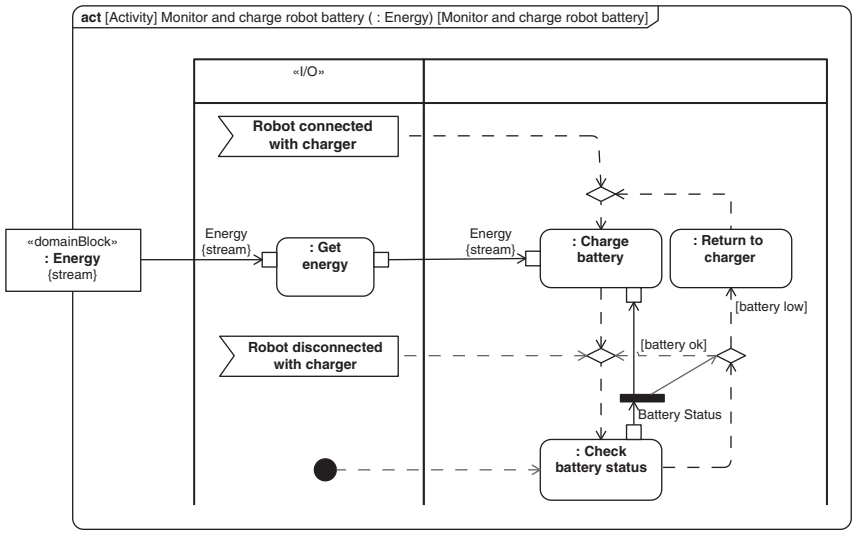


Figure 10.10 Example activity diagram *Monitor and charge robot battery*.

and often closely related to specific technologies than the system behavior. We follow the principle to separate unstable from stable parts (Section 9.5) and separate the use case steps accordingly. We assign them to separate I/O activity partitions as depicted in Figure 10.10. It is also important preparatory work for the FAS method (Chapter 17).

A more detailed description of modeling use cases with activities is provided, for example, by Weilkiens [267, 271].

10.2.5 Model the Domain Knowledge

The domain knowledge represents entities from the systems domain that are known and used by the system. You find the entities in the object flow of the use case activities. The domain entities are the types of the inputs and outputs of the use case activities. The domain knowledge is modeled in SysML with blocks and associations and depicted in a block definition diagram. It is also called a concept model.

Figure 10.11 shows an extract of the Virtual Tour system domain knowledge. The blocks have the stereotype «domainBlock» from the SYSMOD profile [271] to explicitly mark them as blocks of the domain knowledge. Each domain block could have value properties. Associations between domain blocks specify reference properties that have other domain blocks as its type.

For example, the association end “booked tours” specifies that at any time a customer object is always connected with zero to many “Tour Booking”

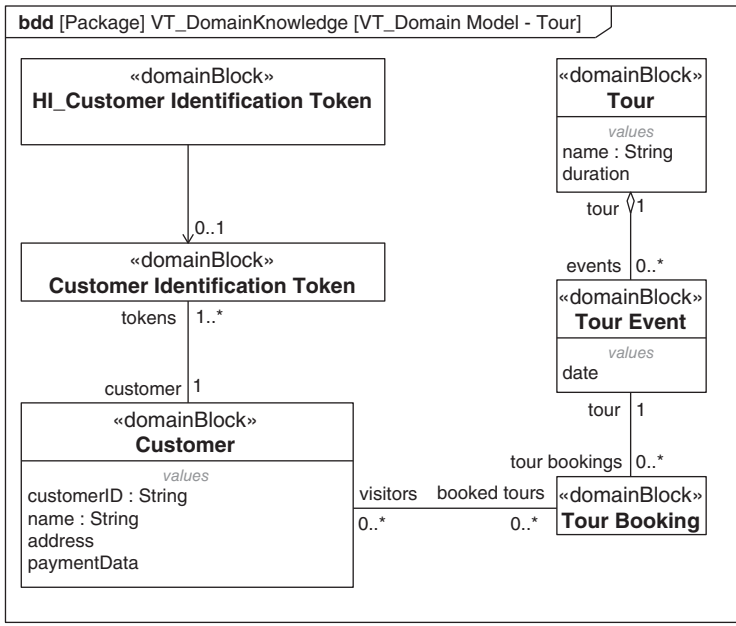


Figure 10.11 Extract of the Virtual Tour domain knowledge model.

objects. The association end specifies a reference property "booked tours:Tour Booking[0..*]" that is owned by the domain block "Customer."

In software-intensive parts of a system, the appropriate domain blocks are closely related to the conceptual data model. However, the domain knowledge does not follow the design rules of a real database model. In addition, the domain knowledge also contains domain blocks that represent real physical things and no data, for example, starlight in a telescope model [125] or "Energy" in Figure 10.10.

The domain blocks are the types of the objects that flow through the use case activities. The separation of the input and output behavior leads to a separate layer in the domain knowledge. In Figure 10.11 the domain block "HI_Customer Identification Token" is the representation of a customer identification token in the context interface layer (see Section 11.5 about layers).

Another part of the domain knowledge is a list of value types and the list of units used in the system model. Most units are common and could ideally be retrieved from a model library, for example, the "ISO 80000" library (Section A.1). New units that are defined for the system-of-interest are part of the domain knowledge and good candidates to be extracted into a separate model library for reuse in other projects of your organization.

Some of the domain blocks, value types, and units of the domain knowledge are also used in the architecture models, for example, as types of interface specifications, value properties, or other architecture elements.

10.3 The SAMS Method

The "Storyboard Activity Modeling for Systems" method (SAMS method) uses storyboards to identification and specification of use cases, including use case activities. The focus here is on designing suitable storyboards, their transformation into use cases, and the creation of traceability between the storyboards and the resulting system models [152].

Storyboards have the benefit of providing system developers with a realistic impression of the system's application. Storyboards are mentioned in the literature as a means of system analysis, for example, in the context of requirements analysis, usability engineering [208], concepts of operations [247], or in similar contexts [235].

Leaving aside the use of storyboards in the film industry and considering their mention in the software or systems analysis literature, they are often found as sequences of sketches of the user interface of software (for example, [40]). However, there are also forms in which one sees the interaction of the user with systems in the real deployment environment. Such a representation can be found in [208]. In [[208], p. 49], explicit reference is made to the possibility showing with storyboards "special or complex environments in which the system is used." Such an approach is very close to that of the "operational scenarios" from the literature in the environment of the architecture frameworks (Chapter 19).

The SAMS method shown below is intended to integrate the use of storyboards as a means of use case analysis into model-based system development. For this purpose, storyboards show a very generalized context for the interactions between a system with its environment. Furthermore, the procedure for the analysis is specified and it is described how traceability can be established between storyboards and other artifacts of model-based system development. The demand for traceability aims to make efficient use of the power of storyboards to make the realistic operational environment of a system observable and discussable at any time, even far away from its actual use. The target is to assign (partial) aspects of storyboards to system model elements. These aspects shall be retrievable in a context-sensitive manner from the system model. This is done using techniques for linking sketches to models, as described in [173, 175], and [12] for sketches in the area of construction. A related idea, namely, the creation of traceability between UML use case diagrams and textual descriptions of stories, was pursued in [247]. The latter

approach will not be considered further here because stories are to be explicitly captured in storyboards using images with the goal of high clarity.

10.3.1 SAMS Method Definitions

A storyboard is a static, content-related sequence of images with the goal of describing the reality of the application [208] of a system or the system's interaction with its environment. Thus, for example, sequences of drawn, photographed, or extracted frames from a movie are considered storyboards if they are statically arranged next to each other. In contrast, a film is not a storyboard according to the definition chosen here because the sequence of images it contains only becomes visible in a dynamic playback process and is thus not understood as a static arrangement. Each individual image in a storyboard is described as a storyboard image. A storyboard image can contain many storyboard image sections which highlight specific parts of the image. In order to establish traceability between the contents of a storyboard on the one hand and the elements of a use case model on the other, storyboard image sections are linked to use case activities.

Figure 10.12 depicts the SAMS method ontology, including how the SAMS method terms are related to the requirement and use case terminology.

Storyboards are part of requirements specifications. Storyboards contain storyboard images which can be further decomposed in storyboard image sections. The whole storyboard image or only a section within the image points to 1 or many use case activities. This link is the bridge from the storyboard into the system

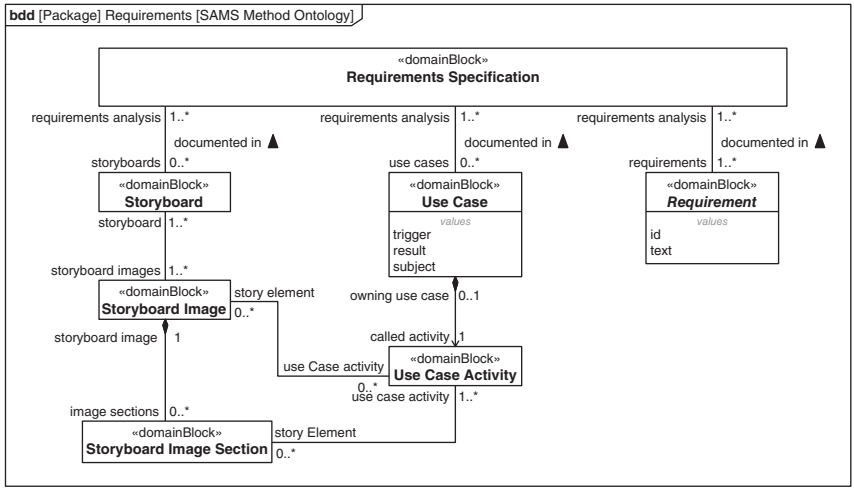


Figure 10.12 SAMS method ontology.

model. Further down, you can read how the bridge can be established in a SysML modeling tool.

10.3.2 SAMS Method

The application of the SAMS method requires storyboards that sharpen the understanding of the system. Therefore, in the opinion of the authors, a proper application of the method requires the collaboration of a person with the necessary artistic talent (hereinafter referred to as the role of "the storyboard designer"). It is assumed in the following that the method is applied in workshops with stakeholders of the system development where the storyboard designer is also present. In principle, other forms of working with the method are also conceivable, but they might require a redesign of the workflows described below.

Like most system analysis techniques, the SAMS method further assumes that the following descriptions about the system are available:

- The definition of the system context, including the system interfaces with human actors and neighboring systems.
- The description of the base architecture (Section 9.2), that means, the solutions for the system, which are taken as given despite the otherwise applicable requirement to analyze as solution-neutral as possible.

Due to the necessity of visualizing the system, solution options may become recognizable in the images of a storyboard without the corresponding solution decisions already having been made. Before using the story board technique, it should therefore be clear to all participants that the solutions appearing in storyboard images are only one possible solution for each design problem and that the other possible solutions are not yet discarded. An exception to this are solutions that are already defined in the base architecture.

The SAMS method begins with agreement on the "stories" to be rendered. Very early in the work on the story, there should be an agreement on the representation of the human actors. The better known the information to be considered about typical actors in the system, the higher the chance of a good analysis result. In order to condense such information, the technique of "personas," which means creating profiles of fictitious persons to represent specific user groups, is suitable [235]. In the context of the visual representation of actors in storyboards, as many aspects of the personas as possible should be captured pictorially. Figure 10.13 illustrates this by showing a possible set of personas for the Virtual Tour system. From left to right: tour guide, virtual tour customer, exhibition planner, and service technician.

Figure 10.13 Virtual tour personas. © 2021 Jakob K., reproduced with permission.

Working on a scenario with the storyboard technique shall only be executed while expecting benefits for the project. It is not meant to have a complete set of storyboards. The work is iterative, that means, the work steps are run through several times, at first still with imperfect states and in the further course with ever more mature result. In advance, the storyboard designer should be made aware of this way of working because a part of the images needs changes in the course of the various iterations. Based on such information, the storyboard designer may decide to use digital rather than analog drawing tools.

First, a story is conceived and submitted to the storyboard designer. He or she then designs the storyboard, and the draft is revised with the participants. A storyboard may also contain elements that are neither part of the system nor of the system context. Only in this way can the operational environment of the system, in general, be made clear without too many restrictions. However, each element shown should have relevance to the understanding required in the system development. When the storyboards are processed further in a model, the elements that are not part of the system context disappear.

While working on the storyboard, one needs to remember that the perfect is the enemy of the good. Not all steps of a scenario can be recorded in detail without compromising the clarity of the storyboard that promotes understanding. The analysis steps that still follow offer the possibility of adding certain details directly to the model later.

While the storyboard designer is still busy finalizing the storyboards that have been defined together, the rest of the team can already start the next analysis step:

In this step, use case models are created based on the storyboards. For scenarios that were not selected for storyboarding from the beginning, this step is also required and starts without the intermediate step of storyboard development. In any case, the result is now a set of use cases as in model-based use case analysis without the SAMS method (for example, [267, 271]). If a storyboard exists, images or image sections are now assigned to the corresponding elements from the use case model (Figure 10.12). How this can be realized in detail depends on the modeling language and the modeling tool used.

Figure 2.2 depicts a storyboard for the virtual tours system. It shows the booking of a tour in a museum and the tour with the robot itself. Figure 10.14 shows one of the images of the storyboard with an image section linked to the appropriate use case activity "Select tour", which is part of the use case "Book a tour." Images and image sections are not supported by SysML. In Figure 10.14, we have used a comment element to model the image section and link it to the activity element. However, the link between the image and the comment is only stored as layout information, and we rely on the feature that the tool supports embedding images in a diagram. You can develop more sophisticated extension plug-ins to integrate storyboards into the model, for example, Albers et al. [12].

With the SAMS method, storyboards can be used for analysis in system development. An artistically talented person should take on the role of storyboard designer. The result is use cases suitable for further work, for example, with the FAS method (Chapter 17). The storyboards can contain aspects of the solution that have not yet been definitively decided as a solution. However, the pictorial representation makes it easy to recognize them as solutions and thus to mentally trace the separation between function and solution. In order to clearly identify aspects of the solution, the corresponding sections of the images could be assigned to elements of a model of the logical architecture that is added ad hoc during the

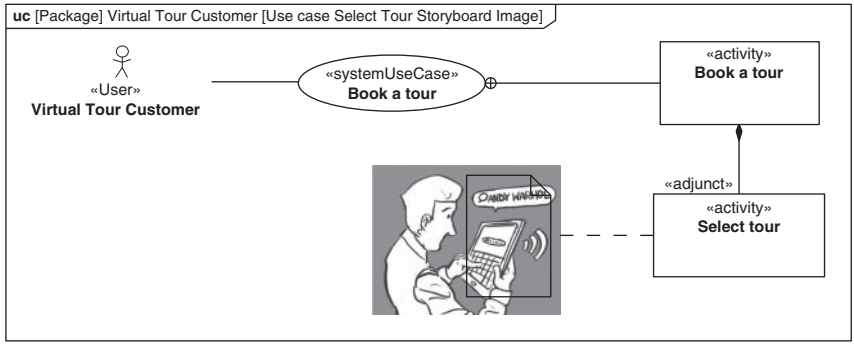


Figure 10.14 Storyboard image "Select tour". Storyboard image © 2020 Jakob K., reproduced with permission.

system analysis. In this way, the identification of implicit solution finding once it has been made would remain explicitly traceable from then on, and elements of the solution that have not yet been identified could be marked in the model.

10.4 Use Cases 2.0

The concept of use cases became famous in the 1990s. It was the time before agile development became famous, although iterative and incremental development approaches were already well known, for example, Boehm [31]. With the success of agile development, Ivar Jacobson et al., the mastermind behind the use case concept, published an updated use case approach in 2011, which he called "Use Case 2.0" [129].

A use case specifies all the ways of using a system to achieve a result. The definition of a use case does not refer to the size in terms of the number of functions or required implementation. Therefore, a use case can be very large and thus not a suitable unit to be implemented in one iteration in terms of agile development. The Use Case 2.0 concept introduces the use case slice that represents only one or some ways of the possible use case performances that can be handled as a unit from specification to implementation within an agile approach (Figure 10.15).

Figure 10.16 depicts the use case activity "Steer the robot - human" of our Virtual Tour system. It is not complete and already shows several "stories" how to perform

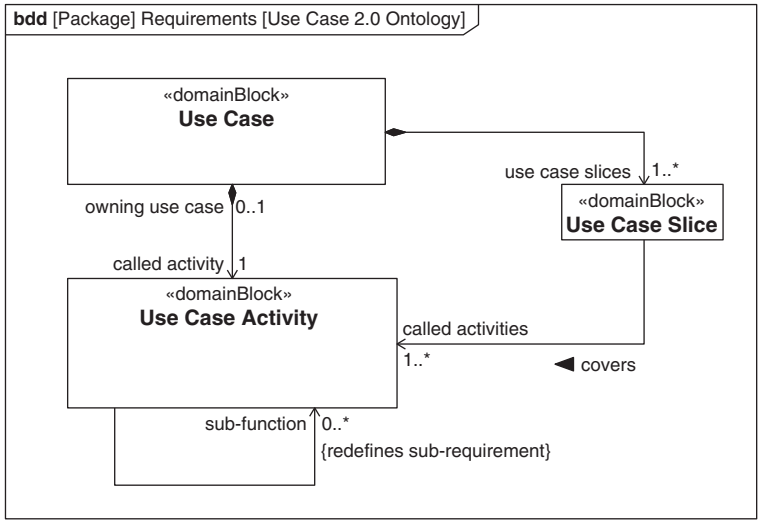


Figure 10.15 Use Case 2.0 Ontology.

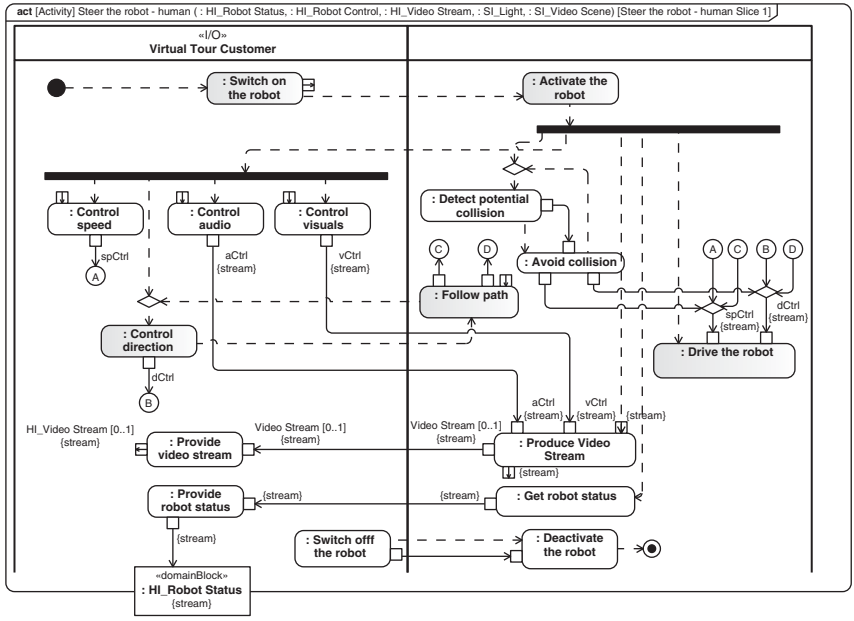


Figure 10.16 Use case activity "Steer the robot - human."

the use case. The greyed steps of the use case are parts of a use case slice, which means a single story within this use case: The virtual tour client switches on the robot, the robot is activated and follows a predefined path.

The next slice adds the collision detection and the avoidance maneuver. The third slice includes the video stream and so forth. Overall, it leads to incremental specification of the use case and derived elements, such as a functional architecture (Chapter 17) to physical architecture and implementation. Each slice represents a complete valuable story.

11

Perspectives, Viewpoints and Views in System Architecture*

11.1 Introduction

The system architect and other architecture stakeholders like to see appropriate architecture views that describe the architecture of the system-of-interest. Views to address different stakeholder concerns are a well-established concept, as already discussed in Chapter 8. For example, let's look at Kruchten's "4 + 1 View Model" [147]: It addresses the problem of architectural representation that overemphasizes certain aspects of development while neglecting others. The "4 + 1" views are the logical view, the development view, the process view, the physical view, and the so-called scenarios,[1] which tie the elements of the other four views together. Kruchten's work was targeted at software architecture. Nevertheless, the notion of separating different stakeholders' concerns via different views also holds for system architecture, as we have seen in Chapter 8.

The term *view* is also known from Section 9.7, which is about the separation of view and model. A view in that sense is a general means of representing only as much of the available information as is practical. Thus, we have to distinguish views in general from the ones addressing a given stakeholder's concern. Since this chapter is about architecture stakeholders, we call the latter kind of views the *architecture views*. Their role in architecture descriptions has been discussed in Chapter 8.

When looking for suitable stakeholder-specific views, we may find a set of different stakeholders who are all interested in the same kind of information about the system, but still have different focus areas. For example, both logistics people and maintenance people may be interested in the geographical spread of the system, but while logistics people may be more interested in weight and maximum

*Together with Matthias Dänzer

1 In the context of this book we would prefer the term *use cases* on *scenarios*.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

dimensions of objects to ship to different locations, the maintenance people may be mainly interested in spare parts needed at different local stocks. Both kinds of stakeholders thus need different views, but they are both interested in the geographical kind of information.

In order to group information of the same kind, we have adopted the notion of *perspectives* from the TRAK architecture framework that will be explained in Section 19.3.7. We can use perspectives as a means of grouping or categorizing architecture views, as we have seen in Chapter 8. The perspective from which both logistics and maintenance people in our example would like to see the system is one whose focus is on geographical spread of the system. However, each of them would like to see a different view of the system, as explained earlier.

Since our definition of architecture views is tied to addressing stakeholders' concerns, we would like to be able to serve architecture stakeholders with accurately filtered information. Therefore, we have chosen to reserve the notion of architecture views for means of presenting narrow subsets of information, hence decomposing representations of e.g. "the physical view" of the system even further. In a model-based approach, it is easy to maintain a multitude of such narrow views, because the model can ensure their consistency.

In this decomposition of what needs to be shown into narrow architecture views, the corresponding architecture viewpoints can ensure that it is clear which information needs to be displayed via a certain architecture view. The architecture viewpoints take the role of defining the expectations toward the model, by defining the questions the model should answer via the architecture views. This makes the architecture viewpoints take a driving role in the modeling, setting criteria that allow to determine whether the model is complete. Well-defined viewpoints may even help detecting over-modeling [89].

Having the possibility to address stakeholders concerns in a very focused ways with these proposed narrow architecture views, we can now exclude the existence of something like "the physical view," because we have so far not encountered any stakeholder who likes to see the complete modeled information about all physical elements in the system at once. We have therefore used the notion of *perspectives* for covering aspects like the "physical" representation of the system. This gives us different perspectives than the ones proposed in TRAK. While these are Enterprise, Concept, Procurement, Solution, and Management, the perspectives we propose are inspired by typical categorizations in the literature like "functional," "physical" (e.g. [145, 253]), "behavior" (e.g. [103]), "layered" (e.g. [163]), or "deployment" (e.g. [213]).

This gives us the *functional perspective*, the *physical perspective*, the *behavioral perspective*, the *layered perspective*, and the *system deployment perspective*. The latter one is derived from the deployment views of software architecting but has been extended here for the sake of systems architecting. It will allow to accommodate

for environmental and spatial information but also for the geographical spread of the system that was mentioned in the above logistics and maintenance example.

We have dedicated a section in this chapter to each of the above perspectives. For example, Section 11.6 will provide more details about the system deployment perspective. Later in this chapter, we will discuss how the different perspectives relate to each other – via yet another perspective that looks at the mapping of entities of different nature. We will also see how such *mapping perspectives* can be used to show the tracing of system elements to other elements we have discussed in this book, like, e.g., requirements.

Each perspective can have its own domain knowledge model according to Section 10.2.5. We will see in Section 11.5 that the domain knowledge model of the layered perspective can be even subdivided by layers.

11.2 The Functional Perspective

Seen from the functional perspective the system is considered as a set of interrelated functions, that means, of different relationships between inputs and outputs of the system itself or elements within it. The entities to be described when looking at the system from the functional perspectives are thus the elements of the functional architecture, i.e. the functional elements (see also Chapter 17).

A very simple and abstract representation of functions is a hierarchical decomposition of system functionality into functions and their sub-functions. Howard Eisner presents a simple informal diagram that can express such a decomposition ([66], p. 146). Figure 11.1 shows how parts of the Virtual Tour system could be functionally decomposed in such a nonformal diagram. The elements shown in that figure are called functional blocks. They could also be modeled more formally in SysML. This is described further below.

Figure 11.1 illustrates an important notion of the functional perspective as it is defined here: the functional blocks are static. This means that functions are represented independent from preconditions needed for making use of them or constraints on the sequence in which they can be provided by the system. These latter aspects would belong to the behavioral perspective (Section 11.4). Therefore, functional blocks are very similar to "system elements" according to Chapter 5, and their interrelations can be seen as system element interactions. As a consequence, interrelations between functions can be modeled by the same means as interfaces between system elements. This will be discussed further in the context of the so-called FAS method in Chapter 17.

The specialty of functional elements is that they may not be realized by individual system elements of the physical system. For example, the functional group "Tour Management" of the Virtual Tour system may be scattered across different

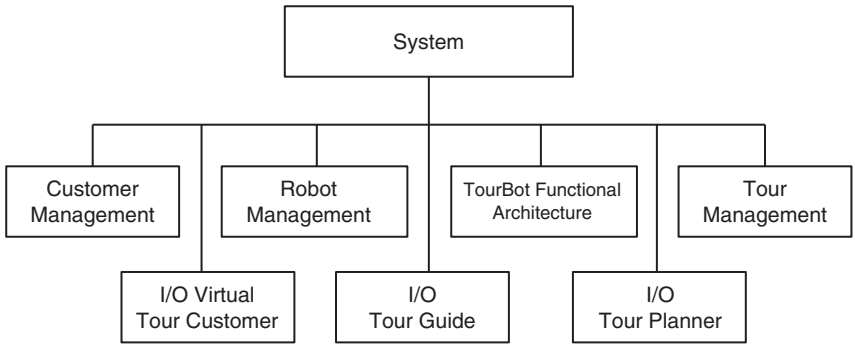


Figure 11.1 Excerpt from the functional decomposition of the Virtual Tour system, presented in an informal way.

servers and client computers or handheld devices in the building hosting the tour itself, in the cloud and within the virtual tour client's reach. People working with views from the functional perspective may need abstract thinking, due to the possible difference between the system as it is seen from the functional perspective and the real system as it is seen when disassembled.

The functional perspective allows for describing the complete system conceptually, even if some implementation details are not yet known. It can provide a high-level overview of the system that does not only enable an understanding of the system's principles of operation, but can also be used to trace fulfillment of functional requirements long before the implementation details are settled. The latter can be exploited to create rough work breakdown structures for covering functional requirements in a very early phase of development. Views of the functional perspective are thus not only interesting for technical stakeholders but potentially for development managers or project leaders as well. Users of the functional perspective should, however, pay attention to the fact that solutions and constraints necessary for satisfying nonfunctional requirements are often underrepresented or even not visible via the functional perspective.

It may be hard to see the value of the functional perspective for people concerned with very detailed engineering problems because it is very abstract. This is one reason why we have seen organizations face difficulty during the introduction of the functional perspective. Indeed, it is more intuitive for people with engineering background to model the system from a physical perspective. In cases in which the functional perspective is omitted, we do, however, see the risk of missing a clear link between functional requirements and system architecture. The resulting observation that something is missing may then lead to making physical models too detailed, resulting in over-modeling or redundancy between the architecture

description and the development documentation produced outside the scope of the system architect.

On the one hand, the functional perspective is thus beneficial for keeping the overview and ensuring an appropriate abstraction level in architecture descriptions; on the other hand, it may be hard to understand or at least hard to accept for parts of the organizations. This is why we recommend to system architects who consider the functional perspective to be valuable to their organization that they declare this perspective as one of the most important ones. Of course, this is dangerous, e.g. regarding the potential underrepresentation of nonfunctional aspect that was already mentioned. Still, the more physical and behavioral perspectives can usually be introduced very intuitively and can just be made without too much need for justification, whereas we have made the experience that the functional perspective needs thorough explanation and should thus be in the focus of the system architect until it is well established.

11.2.1 SysML Modeling of Functional Blocks

To reflect the static nature of functional blocks, we recommend modeling them as blocks in SysML. This way of modeling has been proposed by Lamm and Weilkiens [153, 154] and also by Fernández-Sánchez et al. [76]. The corresponding representation of Figure 11.1 is shown again in Figure 11.2, this time in a SysML block definition diagram.

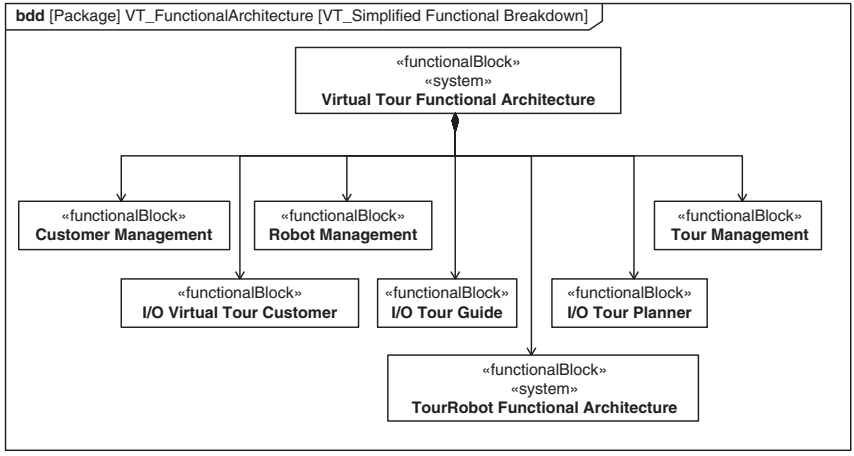


Figure 11.2 Excerpt from the functional decomposition of the Virtual Tour system, in SysML.

The flows of information (signals, data), matter, force, or energy, between different functions can be modeled via ports and connectors. Chapter 17 elaborates on this way of modeling in the context of the so-called FAS method.

11.2.2 Architecture Views for the System Architect

The system architect will need certain architecture views for the own work on the system architecture, for example:

- A set of architecture views showing only excerpts from the functional architecture, for example around a certain feature of the system. One can create a set of them which together covers all functional elements. They can be used e.g. to make a rough assessment on the status of the work on different features of the system.
- An architecture view showing all functional blocks in the system down to a certain hierarchy level. A typical concern of the system architect to be addressed with such a view is an impact analysis that is triggered by a change request. One option could be to generate such views only if needed and to discard them after being used, in order not to have too many details to maintain. As an example, let us assume that the Virtual Tour system would need to be changed in order to comply with a new privacy protection policy of a certain museum that requests computers storing customer data to be locked up in separated rooms with limited access, instead of the common server rooms used so far. Of course, not all components of the system can be locked up in such a special room. A typical procedure for the impact analysis would be:
  1. Generate a list of all functions in the system
  2. Walk through the list systematically and highlight all functions that process customer data
  3. Assess which subsystems contribute to the highlighted functions and find out if a repartitioning is needed to subsystems that can be locked up in a dedicated room
  4. Assess the effort and risk that is related to the repartitioning task

There are many more architecture views that the system architect might like creating either in an ad hoc manner or well planned in advance. The above examples are just a subset, which we saw used in practice.

11.2.3 Different Architecture Views for the Stakeholders of Different Functions

At the beginning of this chapter, we explained that stakeholders who are interested in the same perspective of the system may still like to have different views from

the same perspective. This need is addressed by model-based system architecture, because it enables the creation and maintenance of very focused architecture views for different architecture stakeholders.

The functional perspective can be used to scope development activities that are focused on one function of the system (e.g. [151]). Teams performing the development activities might like to see their own architecture view of the system from the functional perspective. This is typically a view on the functional element in focus of the current development activity, together with those functional elements it has interrelations with. In that case, the system architect can create an architecture view that is constructed around the functional element of interest.

Let us assume that a certain team in the development of the Virtual Tour system has the task to implement the tracking of the different robots' utilization at one site – for example, if the robots are available and at what position they are. Such a team could have a functional block "Robot Utilization Tracker" as its scope, but it would also like to know how to obtain position information and which other functions in the system would need that information. An architecture view for such a team is shown as an example in Figure 11.3: A SysML internal block diagram shows excerpts from the functional architecture. The functional element of interest is represented by a block symbol with inverted colors (black fill color, white font) to distinguish it from the other elements. Furthermore, the figure shows the direct interrelations to other functional elements. Elements without such direct interrelations to the element of interest are omitted in the diagram.

In the given example, it is possible to make a formal definition of the contents to be displayed in such a view, like "the block of interest and the blocks that have interfaces via ports with it." If a high number of views according to the same definition needs to be maintained for different functional blocks of interest, then automation features of the modeling tool can be helpful: An automatic view management tool could be implemented. It would automatically create the needed architecture views – or at least, it would check whether existing architecture views comply to the viewpoint definition and indeed show all model elements that the definition requires to be shown. An example of such a tool will be presented in Section 11.11.3.

11.3 The Physical Perspective

Seen from the physical perspective, the system consists of system elements that can exist in the real world. They are thus the elements of the physical architecture, if they are not so detailed that they would rather belong to the system design or even outside the scope of the system architect. These elements' interconnecting interfaces will also be in scope of the physical perspective. Therefore, interface

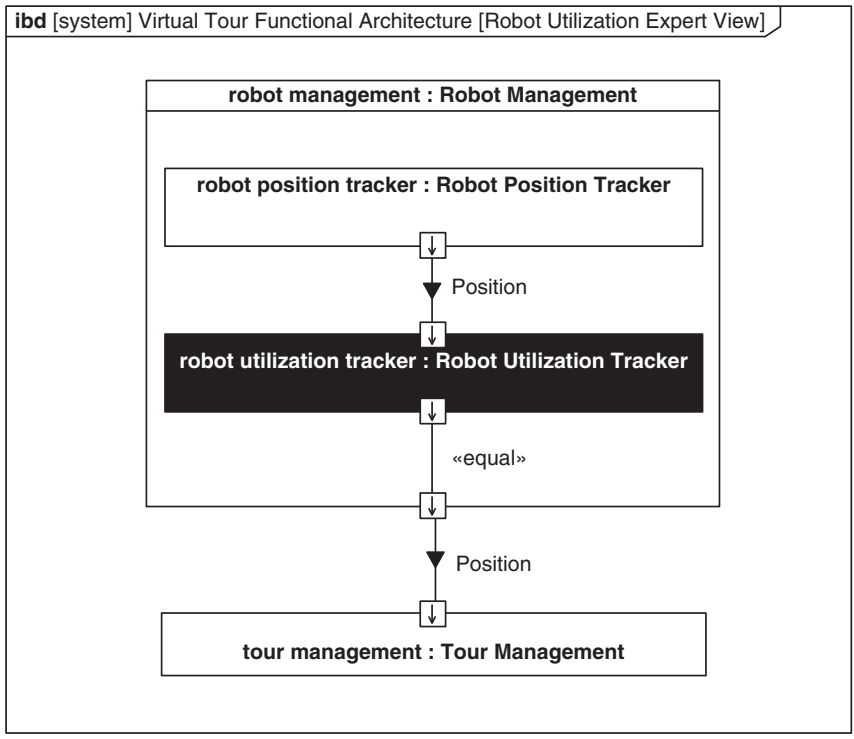

ibd [system] Virtual Tour Functional Architecture [Robot Utilization Expert View]
robot management : Robot Management
robot position tracker : Robot Position Tracker
Position
robot utilization tracker : Robot Utilization Tracker
«equal»
Position
tour management : Tour Management

Figure 11.3 An architecture view from the functional perspective for a functional team working on utilization of robots, highlighting the team's scope with inverted coloring of the corresponding element.

control documents (e.g. [145]) may also be assigned to the physical perspective, possibly more under the headline "system design," than under the headline "system architecture."

One can look at multiple levels of abstraction from the physical perspective. We saw in Section 9.1 that system architecture may comprise the base architecture level, the logical architecture level, and the product architecture level. From the physical perspective, one can focus on any of these levels.

11.3.1 Logical Architecture Example

The logical architecture is a representation of the system in which the system elements are defined according to the technical concept for implementing the system, but not necessarily enough concrete for implementing the system. They are the

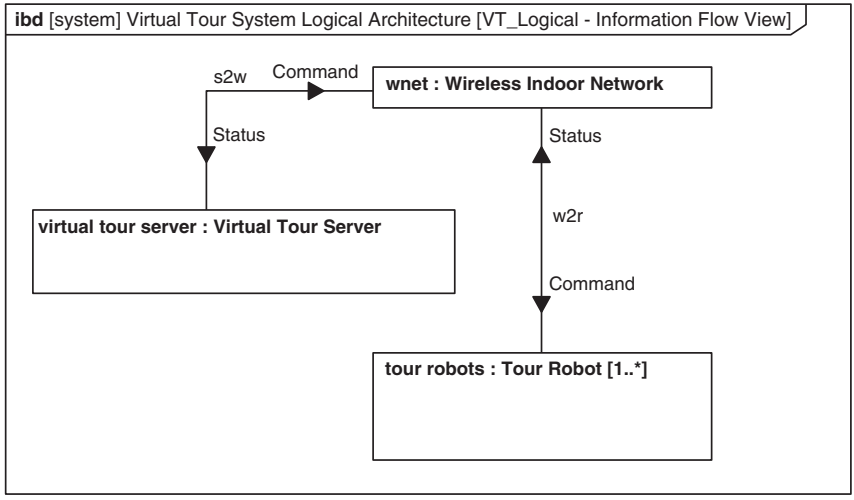


Figure 11.4 Example architecture view on the Virtual Tour system in the logical architecture.

elements of the logical architecture. An example will be provided in the following. See Chapter 5 for the general definition of the term "logical architecture".

Figure 11.4 shows an example, which is related to the Virtual Tour system: The "Wireless Indoor Network" is a solution for the communication inside the facility covered by the virtual tour. The technical concept is to use a wireless network solution instead of a proprietary point-to-point radio protocol for communicating with the robots. The logical architecture does not define which wireless network standard the network will comply to and how the routing of information inside the building will happen.

Interfaces can be defined roughly in the logical architecture. To give an example, Figure 11.5 shows some data details of the interfaces from Figure 11.4 by defining a domain knowledge model in a SysML block definition diagram. This kind of modeling can support the specification of interfaces (here: from the logical perspective).

11.3.2 Product Architecture Example

The product architecture is concerned with the realization of the solution with tangible elements of the real world (see also the definition in Chapter 5). To continue the example from Section 11.3.1, the product architecture is thus about the concrete network technology used to establish indoor communication between the server on site and the robots.

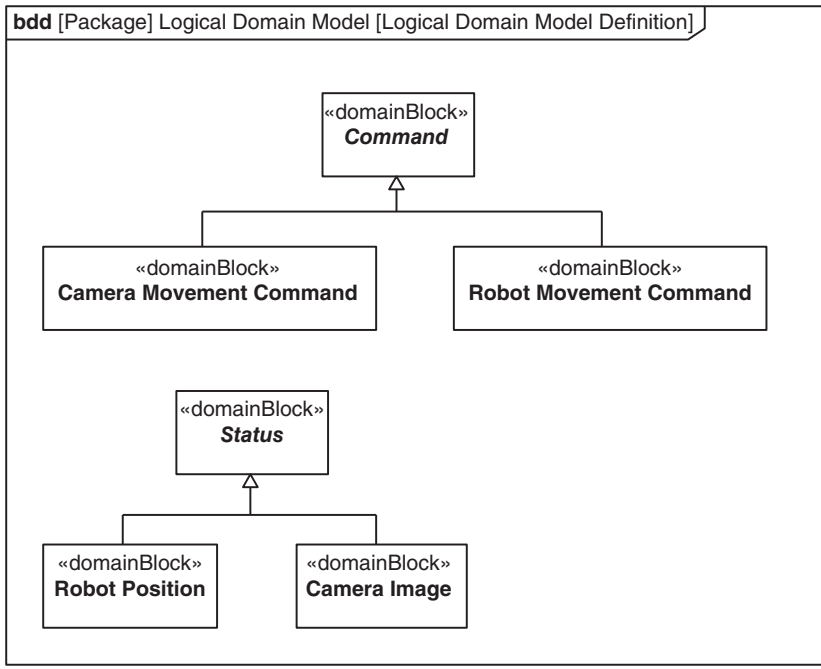


Figure 11.5 A data view related to the logical architecture as the basis of interface specifications: needed details are defined by further specifying the data flows from Figure 11.4 in a domain knowledge model.

Let us consider that the wireless indoor network is established by placing wireless access points that provide a wireless network across the whole facility to cover by a virtual tour. The access points are connected to the local server via a powerline network,[2] in order to save cost for the installation of a wired network connecting them with the server. This means that both the server and the wireless access points need connections to the facility's electrical network in order to be provided with both power supply and access to the indoor network.

The concrete solution "powerline network" will typically become evident via the physical perspective, whereas the logical perspective has just shown

2 Powerline networks use existing electrical networks in order to transmit data packages. The user can plug them into existing electrical outlets. They will contact each other via a high-frequency carrier signal transmitted on the electrical power network and establish a local area network for computers or similar devices. Here we choose this technology to illustrate that views are non-orthogonal: the electrical and communication aspects of the system overlap due to communicating via the electrical network. The consequences of non-orthogonal views will be elaborated on in Section 14.1.4.

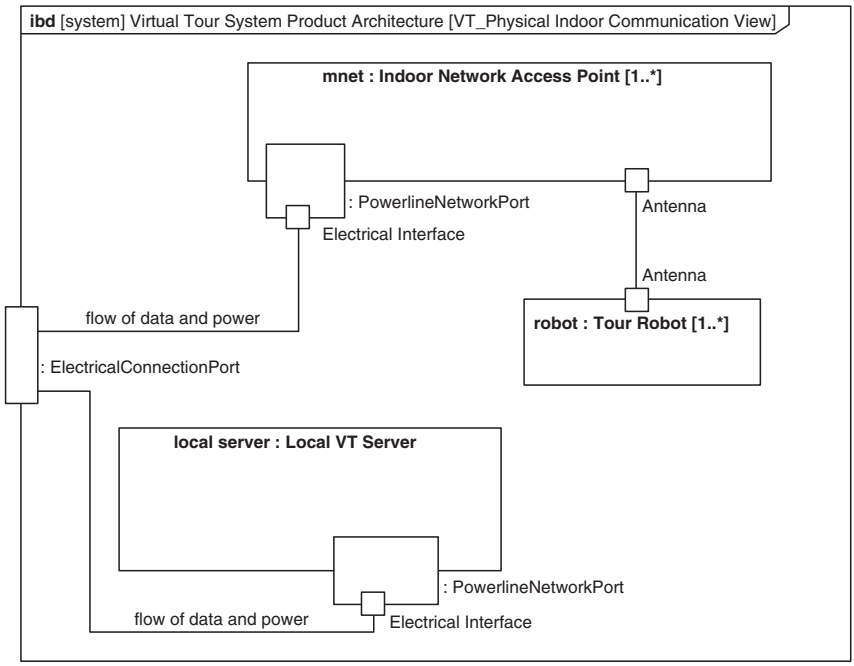


Figure 11.6 Example architecture view on the Virtual Tour system from the physical perspective with focus on communication.

communication flows, as we saw in Figure 11.4. Some architecture stakeholders may also like to see communication flows from the physical perspective. They can be provided with an architecture view like the one in Figure 11.6, where the server and the network access points and some of their connections are shown, with a focus on those connections needed for communication only.

An architecture stakeholder who has to care about installation procedures and installation cost of the system may not be interested in the communication inside the system, but rather in the number of mechanical and electrical connections. This kind of stakeholder may be provided with an architecture view very similar to the previously discussed one, but now with an additional focus on the mechanical aspects of connections. An example is shown in Figure 11.7 in a SysML internal block diagram. Note that in comparison to Figure 11.6 the wireless connection with the robot is no longer shown, because it is neither an electrical connection nor a mechanical connection. Later, we will see how the system deployment perspective may provide additional facts that are relevant for the installation of the system in terms of wireless network performance (Section 11.6).

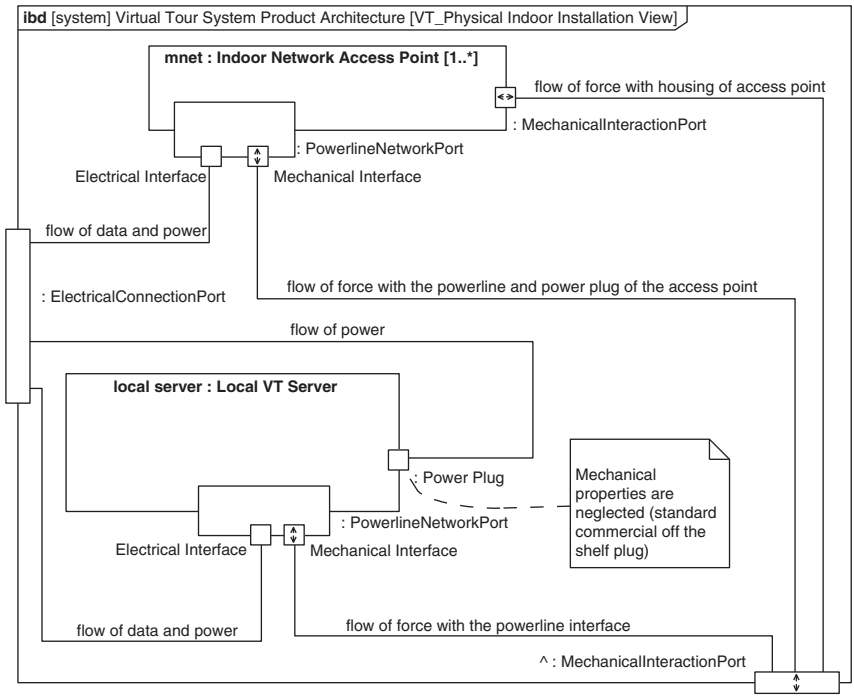

ibd [system] Virtual Tour System Product Architecture [VT_Physical Indoor Installation View]
mnet : Indoor Network Access Point [1..*]
flow of force with housing of access point
: MechanicalInteractionPort
: PowerlineNetworkPort
Electrical Interface
Mechanical Interface
flow of data and power
: ElectricalConnectionPort
flow of force with the powerline and power plug of the access point
flow of power
local server : Local VT Server
: Power Plug
Mechanical properties are neglected (standard commercial off the shelf plug)
: PowerlineNetworkPort
Electrical Interface
Mechanical Interface
flow of data and power
flow of force with the powerline interface
^ : MechanicalInteractionPort

Figure 11.7 Example architecture view on the Virtual Tour system from the physical perspective with focus on installation.

11.4 The Behavioral Perspective

The behavioral perspective defines what the system shall do as a sequence of actions over time. It is concerned with state transitions in the system and the timing and sequence of function calls or information exchange via interfaces. SysML state machine diagrams, sequence diagrams, and activity diagrams with control flows usually show views from the behavioral perspective. The appendix contains examples for such diagrams.

Particularly, software engineers often ask for behavioral specifications of the system, because state machines and programmed sequences of events are usually realized via software or embedded software inside the system.

11.5 The Layered Perspective

11.5.1 The Layered Approach

The layered approach originates from software architecting, but as we will explain, there are good reasons to apply it to systems architecting as well. After a short

discussions of layers in software architecture, we will generalize the idea of layered abstractions to systems architecting for systems with different information processing components.

The layered approach in software architecting is based on separation of concerns. It can be seen as a modern form of a modularization principle formulated by Parnas [198], stating that modularization should aim at defining modules in a way that each module hides a difficult design decision or a design decision that is likely to change. The benefit is that interfaces with other modules do not need to change if a design decision about the innards of a module changes. In other words, the layered approach results in modules with low coupling (see also Section 9.3).

In a layered approach, the modules are assumed to be "stacked" in a certain order and are therefore called *layers*. Indeed, a set of several connected layers can be called a "layer stack." The notion is that a layer can only have an interface with an adjacent layer such that information has to be systematically transformed to slices of the stack one-by-one instead of being able to bypass parts of it. A typical advantage of such an approach is to avoid uncontrolled dependencies between different parts of the system and to enable changing parts of the solution without running into a high risk of having an impact on other areas in the system. This enables flexibility in choosing and changing the concrete technological solution for the implementation. In cases in which flexibility is a dominant quality criterion, the different layers can be chosen to represent different abstractions of implementation details. In that case, layers are stacked in order of ascending abstraction level.

One example of a layer that both hides a design decision and enables the flexible exchange of a solution is a "hardware abstraction layer" in a software product. It is designed to offer a standardized interface for using hardware toward the adjacent layer that is higher up in the layer stack. This means that the upper layers of the stack can use hardware, independent of its concrete realization. The concrete information exchange with the hardware is hidden from the upper layers. This enables porting of the software product to a completely different hardware without changing anything in the implementation of the upper layers. Only the hardware abstraction layer needs to be adapted to the new hardware, but it will still offer the same interface toward the upper layers. In a variation of this approach, the software product can dynamically create different instances of the hardware abstraction layer, each being able to handle a different hardware solution. Dänzer et al. [62] have provided an example from the hearing care field, where a layered approach was used in a software product that interoperates with diagnostic devices from different manufacturers. The layered approach was chosen with the aim to enable an easy integration of the different manufacturers' devices.

All the previously said can be handled by software architects, as long as the layered approach is used inside one software product or inside one software subsystem of a system. So why should systems architecting be concerned with layered architectures? That is the topic of Section 11.5.2.

11.5.2 The Layered Perspective in Systems Architecting

An early example of a layered approach that can be applied to partially non-software systems is the OSI reference model according to ISO/IEC 7498-1:1984 [112], which has been replaced by a new revision in the year 1994. It defines a standardized way of interconnecting systems, based on the notion that higher layers can be specified independent of the technology used in lower layers. Two applications on different devices can exchange information, without a dependency on the carrier technology, so, for example, no matter whether the information is transferred via a cable, a fiber-optic connection or a radio transmission. The lowest layer of the OSI reference model is concerned with the physical transmission technology, which would imply the use of electronic or optoelectronic hardware in the given examples. The OSI reference model thus enables the interconnection of devices with considerable non-software technology within one system. The paradigm in interconnecting these different nodes is to enable their information exchange without the need to develop all of them for the same information interchange medium.

This leads us to one advantage of the layered approach for systems architecting: If the whole system is based on one shared layer model, then different entities inside the system can exchange information without being sensitive to changes of the solution in other areas of the system. An example based on the Virtual Tour system may underline this: Assume that a user of the system has opened the application for controlling the tour robot on a mobile device, in parallel with a geo information system that can display satellite pictures and location information based on the longitude and latitude of a position on earth. The user may now like to see where on the satellite picture the robot's current location is. If the robot position can be retrieved in global coordinates, then this is possible. It may thus seem practical to handle robot locations as global position information as soon as the information needs to be exchanged with other devices or needs to be presented to the user. On the other hand, the controllers that process the information from distance sensors on the robot may not have all the information necessary to convert measured distances into global positions. When organizing the Virtual Tour System according to a layered approach, a higher layer in the layer stack might thus represent robot positions in global coordinates, whereas a lower layer may perceive position information as a distance value. This is shown in Figure 11.8, with a strong simplification: It is not shown that the communication between the robot and the VT App needs some calls to lower layers, in order to use a communication service. A view without this simplification will be provided in Section 11.5.5.

If a whole system has to be designed according to a common layered approach, then it is not sufficient to define the layered architecture as a part of the software architecture. Heterogeneous systems may be composed of very different kinds of

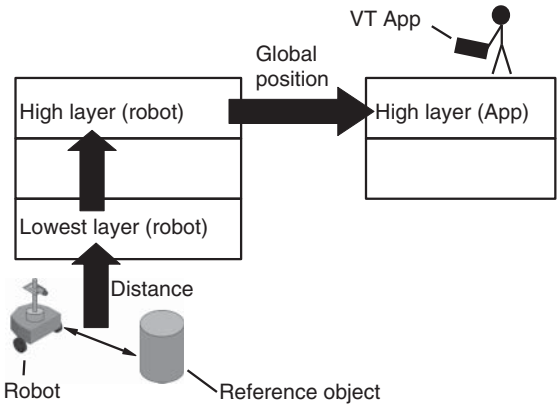


Figure 11.8 An informal architecture view from the layered perspective.

information processing subsystems, involving different information processing components with different architectures. Still it may be desirable to impose a system-wide layer model on all the subsystems. This has to be done via an artifact that is visible across the different stakeholders involved in creating the different subsystems, so best by the system architecture description. The *layered perspective* will be the one with the views needed to impose a common layer model on the system and make it visible to different stakeholders.

The layered perspective has one fundamental difference from the functional and the physical one: While the latter ones are based on a decomposition approach, the layered perspective is based on a service approach. The physical perspective for example allows us to show how a system is decomposed into subsystems, which are part of the system. In a layered approach, a lower layer is not part of a higher layer, it rather offers a service to it: Maier [163] states that layers have an "is-used-by" relationship as opposed to the "is-a-part-of" relationship in decomposition approaches. Maier also points out the challenges in handling layered approaches in systems architecting, which traditionally based on decomposition approaches. Yet we propose to consider the layered architecture as a special physical architecture, because it will contain building blocks that can be found back in the implementation of the system.

Due to this special nature of layered perspectives and layered architectures, a systems architecting approach based on layered architectures should be considered a challenging one, and to our knowledge, there has until recently not been much interest in using a layered perspective like we propose it in systems architecting. This is why we would like to keep the challenges of the approach manageable and therefore define that the scope of layered perspectives is limited to information processing aspects of the system. As a consequence, an important aspect in layered architecting will be to translate information flows from the domain knowledge model (Section 10.2.5) into real-world representation of the information. This will

be discussed further in Section 11.5.3. As a rule of thumb, this means that system elements which are not needed for handling information objects from the domain knowledge model are very likely to be out of scope in the layered perspective.

A central notion of layered architectures is the *information hiding*, this means: information that may be present on one layer may not be available on another layer. As a consequence, information that is highly dependent on the solution or design decisions in certain areas of the system is prevented from accidentally being used in other areas of the system. This helps avoiding side effects and undesirable bad dependencies. In the tour robot example, the exchange of the distance sensor technology may lead to a different representation of distances. The technology-dependent representation should be encapsulated inside the driver layer, such that it is hidden from other layers. As long as only an isolated part inside that layer processes the sensor output, it is easy to analyze the impact of a changed sensor technology: only the part of the layer that uses the sensor output needs to be redesigned (see also the pattern "separate stable from unstable parts" from Section 9.5). The interfaces to other layers will remain the same, and in consequence, only the driver layer needs to be updated. In a solution without information hiding, the output of the distance sensor can in theory have been used by a developer working on an area of the system whose relation to the distance sensor is not directly obvious. This comes with the risk of forgetting this area of the system when analyzing the impact of a changed sensor technology. Information hiding can thus facilitate change management and in particular the maintenance of the system architecture.

11.5.3 Relation to the Domain Knowledge Model

The information hiding can imply that objects from the domain knowledge model may not be available on certain layers. We have already discussed the example of the robot position, which has different representations on different layers. To model the different representation of the same kind of information on different layers, we propose to extend the domain knowledge model with layer-specific domain objects. While the domain object was so far an artifact of requirements engineering, the proposed additional blocks are artifacts of systems architecting. These can then be traced to the corresponding domain object of the original domain knowledge model from requirements and use case analysis. Figure 11.9 shows this, based on the mentioned example: The "Position" from the original domain knowledge model represents some kind of position information to be processed by the system, based on findings during requirements analysis that the system needs to process positions. The layer-specific representations of position information are as follows:

- "Global Position," the kind of position information to be used on the application layer and to also be displayed to the users of the system

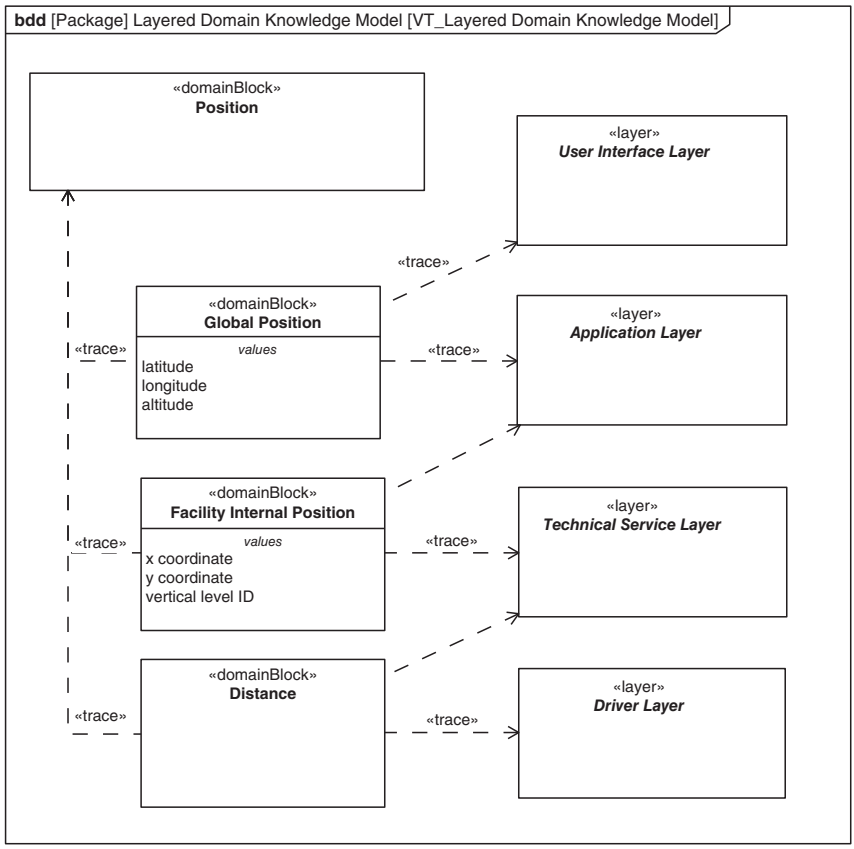


Figure 11.9 Layer-independent domain block "Position" and the derived layer-specific domain blocks.

- "Distance," the kind of position-related information to be used in system elements close to the distance sensors
- "Facility Internal Position," the position compared to one reference point at the facility, to be used on an intermediate layer.

The blocks on the right-hand side in Figure 11.9 represent the different layers. In the shown example, they have been chosen with the aim of separating different abstraction levels. Their modeling will be further discussed in Section 11.5.5. The «trace» relationships from the domain blocks to the «layer» blocks assigns each domain block to a layer. The dependency relationships that point diagonally up to the right indicate that the domain blocks on a given abstraction level are also used on the higher adjacent layer. This is because the information is handed up through the layer stack and needs to be transformed before being in the

proprietary information of the higher layer. Systems architecting will typically handle every «domainBlock» object that is traced from a «layer» object, whereas layer-independent blocks like "Position" in Figure 11.9 are typically artifacts from the requirements and use case analysis.

11.5.4 Architecting the Layers

When defining a layered architecture, one needs criteria on how to separate different aspects into different layers. Here are some considerations:

- We already saw that Parnas [198] recommends hiding difficult design decisions or those that are likely to change. The latter is related to the pattern "separate stable from unstable parts" from Section 9.5.
- It should be possible to release layers separately. This can save reverification effort after changes because the similarity with a prior version of the system is given for those layers that were not updated during a change.
- Layers should be separately testable (sometimes when combined with at least a part of the layer stack underneath or an emulator thereof). System architects together with the appropriate verification stakeholders can assess whether the following scenarios are easily feasible with the chosen layered architecture, in case these scenarios are desirable:
  - Hardware together with its driver layer can be tested by means of test software running on top of the driver layer, without the need for a human user to operate the system. This enables hardware testing before the completion of software.
  - Conversely, the lower layers can be replaced by a hardware emulator, enabling testing of user interfaces or application logic even before the hardware is available.
- It should be possible to generate test inputs for domain objects that are handed over via interfaces between layers.
- It should be assessed whether testability is given with the given domain objects as an input. If the layer is not well-testable as it is foreseen, then this may indicate that the information hiding is not established sufficiently well.

11.5.5 SysML Modeling of Layers

We will explain an approach for modeling layers in SysML. It has been based on a proposal by Dänzer et al. [61]. We propose modeling abstract blocks to represent layers. Concrete blocks are used as types for the part properties of the layered architecture. They can be modeled as specializations of the abstract blocks, which makes them inherit, e.g., the ports from those.

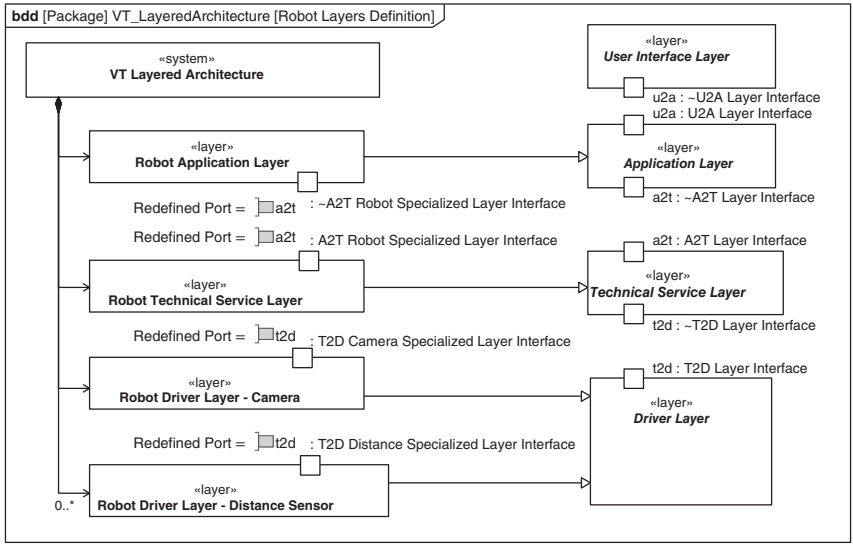


Figure 11.10 Example architecture view on the Virtual Tour system from the layered perspective.

Figure 11.10 shows the definition of layers for the Virtual Tour system. On the left-hand side, the composition of different layered parts into a whole is modeled. The picture shows an incomplete example with just the blocks necessary for the explanation of the modeling approach. On the right-hand side of the figure, the different layers are modeled as abstract blocks with the stereotype «layer». Concrete blocks can be assigned to a given layer by means of a generalization relationship that points from a block to the abstract block representing the layer (so from left to right, in the example in Figure 11.10).

These concrete blocks represent layered elements which can be implemented in a physical subsystem. Their assigned stereotype is «layer». In the figure, we see that there are two concrete blocks of the "Driver Layer" kind: the "Robot Driver Layer – Camera" and the "Robot Driver Layer – Distance Sensor."

Via the generalization relationships, the concrete blocks (on the left) inherit all ports from the abstract blocks representing the layers (on the right). This is not visible in the figure. However, one can see that some of the inherited ports are redefined into more specialized ports. The approach of modeling ports for the whole layer and then redefining them where appropriate ensures that there is a standardized way of interfacing layers to each other. The standardized way of interfacing makes it easier to keep control of interface behaviors, which may, e.g., make it easier to avoid undesired effects during all the different interactions that can occur in

different system configurations. The naming of the ports will be discussed further below.

There are only few blocks in the example given by Figure 11.10, but in the layered architecture we present in the following, there will be more. Still, each concrete «layer» block will be derived from exactly one of the abstract blocks that have already been introduced. This ensures that the different subsystems in the system adhere to the system-wide valid layer split. Here, the different layers have been chosen to separate different abstraction levels. This means that all parts of the system with the given layers are based on a common way of abstraction and can therefore easily be interconnected.

The common way of abstraction and the standardized behavior of a layer across the system mean that the developers of the different parts of the system have a common basis for structuring their work products, which can make it easier for them to understand each other's area of development and communicate with each other, just in analogy to the easier communication between layered elements in the system.

Figure 11.11 also shows how the ports for information exchange between adjacent layers are defined: They are named according to the layer boundary on which they reside. For example, ports residing on the boundary between the user interface layer and the application layer will be identified using the abbreviation "u2a."

Coming back to the robot position example, there are different operations in relation to the robot position. Whether a certain operation makes sense, depends on the abstraction level and hence on the vertical anchoring in the layer stack. The operations are therefore directly related to the domain knowledge model we saw earlier in Figure 11.9: While something of type "Distance" from the domain knowledge model is returned from a "GetDistance()" operation on an interface that is provided by the Driver Layer, something of type "Global Position" is returned from the corresponding operation on an interface that is provided by the application layer. Information hiding is in place and ensures increasing abstraction level from bottom to top of Figures 11.9 and 11.10. It should be noted though that the vertical order of blocks in the diagrams has been made arbitrarily during diagram layout. It is not part of the model but only expressed in the given diagram.

The order of layers is however implicitly given in model. It is implied by the layers' interconnections in the internal block diagram, because layers will only be adjacent to each other when they have a connection with each other. An example of such an internal block diagram with layer interconnections is shown in Figure 11.12: We see parts of the layer stacks of the server application and of the robot at the facility. Again some details are omitted. For example, it is very likely that a server application has a user interface layer, but within the view according to Figure 11.12 this is not shown.

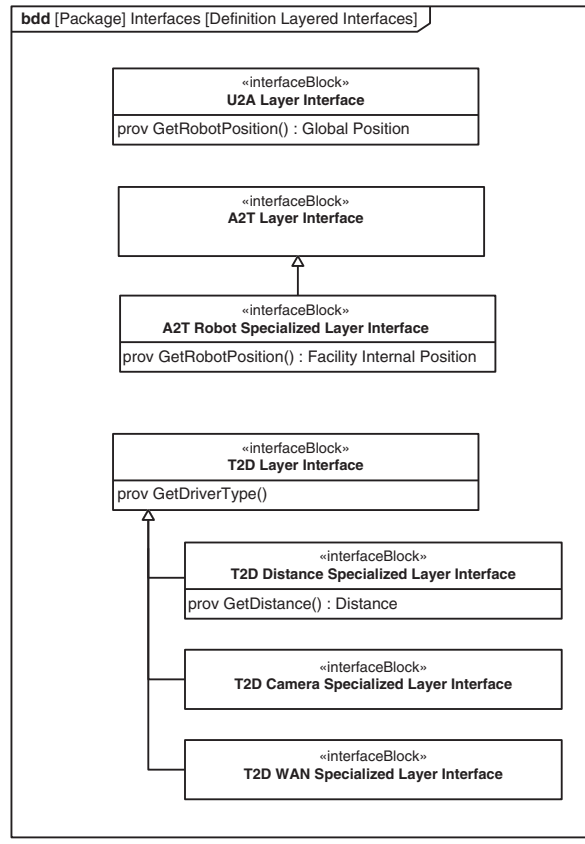


Figure 11.11 An interface view from the layered perspective: details on interfaces are defined by specifying the port types of ports from Figure 11.10 in a block definition diagram.

We have now seen a way to use SysML for a representation like it was previously sketched in the informal drawing in Figure 11.8.

Since the informal drawing was made to explain the notion of information hiding, it does not show the communication between the robot and the user's VT App in detail. However, the formal SysML representation in Figure 11.12 now shows this detail: A "Local Network Driver Layer" on both the server side and the robot ensures that the server and the robot can exchange information via a local network. The network connection itself can be omitted in a layered architecture and deferred to, e.g., the product architecture. In order to still enable the reader of the diagram to follow the information flow, a connection has been modeled between "LAN Server: Local Network Driver Layer" and "LAN Robot: Local Network Driver Layer." This connection has no precise technical meaning in the context of the layered perspective, but provides the reader of the diagram with information needed to understand communication flows.

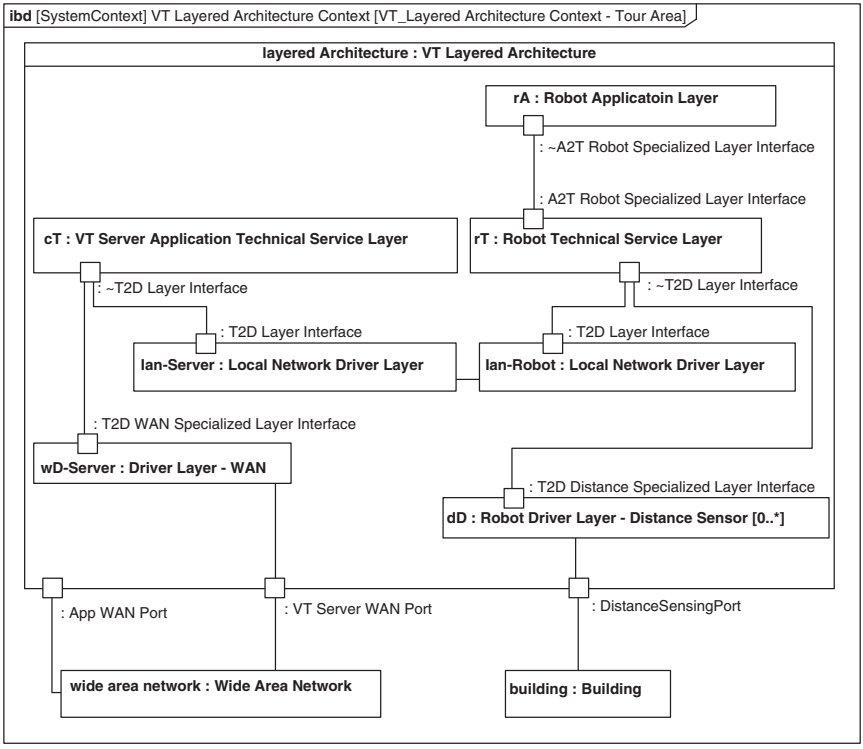

ibd [SystemContext] VT Layered Architecture Context [VT_Layered Architecture Context - Tour Area]
layered Architecture : VT Layered Architecture
rA : Robot Applicatoin Layer
: ~A2T Robot Specialized Layer Interface
: A2T Robot Specialized Layer Interface
cT : VT Server Application Technical Service Layer
rT : Robot Technical Service Layer
: ~T2D Layer Interface
: ~T2D Layer Interface
: T2D Layer Interface
: T2D Layer Interface
lan-Server : Local Network Driver Layer
lan-Robot : Local Network Driver Layer
: T2D WAN Specialized Layer Interface
wD-Server : Driver Layer - WAN
: T2D Distance Specialized Layer Interface
dD : Robot Driver Layer - Distance Sensor [0..*]
: App WAN Port
: VT Server WAN Port
: DistanceSensingPort
wide area network : Wide Area Network
building : Building

Figure 11.12 SysML representation of the layer stacks on different devices, as sketched in Figure 11.8.

In the diagrams shown here, all connections that connect to layers without ports are such cases in which an information flow has been visualized in an informal way. The connections representing context interactions by contrast have a formal technical meaning. They represent information exchange between the system and external elements. In this case, the "Wide Area Network" is one of them. We see that the server can access it via the corresponding driver layer. The Wide Area Network (WAN) is outside the system, but it transmits information to another element inside the system, i.e. the VT App. Figure 11.12 only shows a second port that is connected with the WAN, but the figure would have been overloaded had we shown the layers of the VT App as well. Therefore, a second Figure 11.13 has been created to show that aspect.

By showing the connection of the elements in the driver layer with the WAN ports, the context interactions of the system can be explained from the layered perspective. This was a reason not omit the "Wide Area Network" in the example. By contrast, the "Local Area Network" could be omitted due to the earlier-mentioned

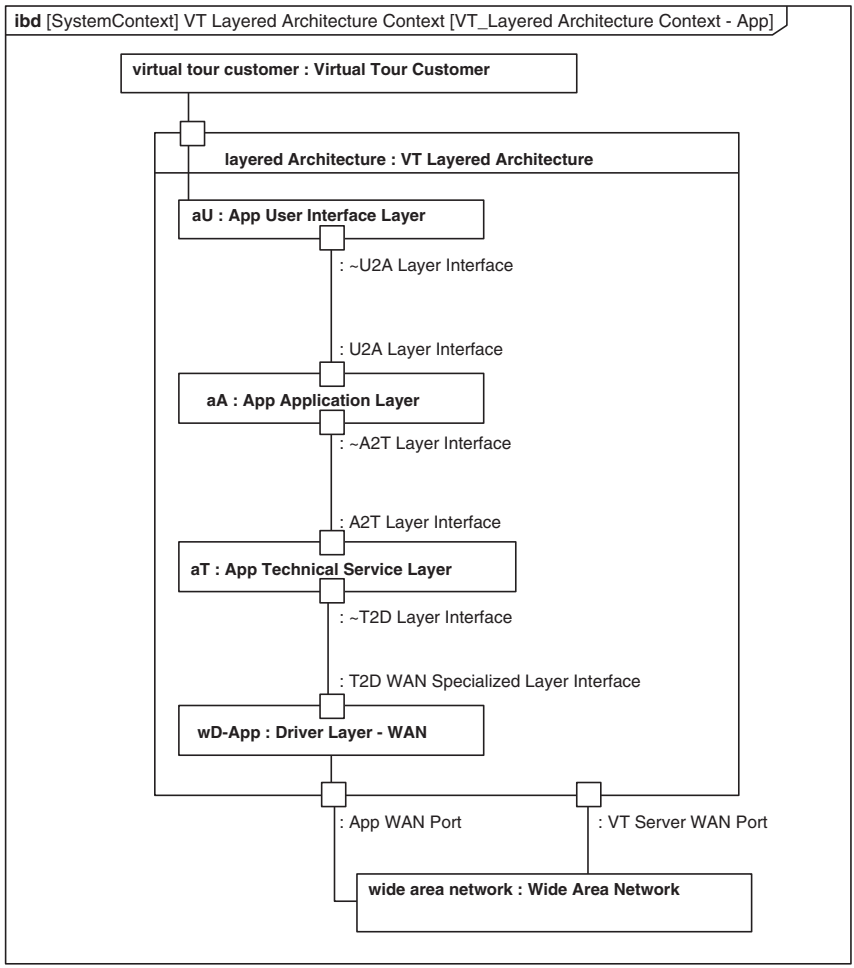

ibd [SystemContext] VT Layered Architecture Context [VT_Layered Architecture Context - App]
virtual tour customer : Virtual Tour Customer
layered Architecture : VT Layered Architecture
aU : App User Interface Layer
: ~U2A Layer Interface
: U2A Layer Interface
aA : App Application Layer
: ~A2T Layer Interface
: A2T Layer Interface
aT : App Technical Service Layer
: ~T2D Layer Interface
: T2D WAN Specialized Layer Interface
wD-App : Driver Layer - WAN
: App WAN Port
: VT Server WAN Port
wide area network : Wide Area Network

Figure 11.13 SysML representation of the layer stacks on different devices, as sketched in Figure 11.8.

simplification in showing the communication flow (direct informal connection between driver layers). Omitting the "Local Area Network" in this case avoids that the system's innards are represented by a mixture of layered elements and elements of the product architecture.

When using the modeling approach according to [61] with SysML 1.3 or later, the interfaces between ports of «layer» blocks should be proxy ports (see SysML reference in Appendix A), because the implementation of operations is provided by entities inside the layers. In the figures shown in this here, we have omitted the «proxy» stereotype to avoid cluttering the diagrams.

11.6 System Deployment Perspective

In software architecting, deployment describes how software artifacts are distributed across the nodes of the runtime environment. These software artifacts could be the outcomes of the software build process, configuration files, execution environments like virtual machines or application servers, libraries, and frameworks. In the case of an embedded software for a multi-processor device, a view of the deployment perspective could describe how the different software modules are distributed to different processor cores. How can this perspective be generalized to be applicable for systems architecting? Kossiakoff and Sweet [145] see deployment mainly as the transportation to the operational site and the subsequent installation. In analogy to software architecture, we would like the deployment perspective to be more static in the sense that it describes the system after it has been deployed. Figure 11.14 shows an example architecture view of the Virtual Tour system from our more static kind of deployment perspective, which we call the *system deployment perspective* from now on, in order to make a clear distinction from the software deployment perspective in software architecting.

Considerations like transportation can still be derived in views of the deployment perspective, because a picture of the deployed system should allow considerations about how to deploy it. In the example in Figure 11.14, one could, e.g., immediately see that a robot needs to be transported to the facility and lifted to a certain floor. Hence, the logistics people from the example in the beginning of this chapter would probably see their concerns addressed with an architecture view from the system deployment perspective, like shown in Figure 11.14.

Considerations about the environment can also be made from the system deployment perspective. One needs to be careful with the term environment though: In a software development context, Rozanski and Woods [213] emphasize the aspect of the system's environment in the deployment context, but of course the addressed topic in their case is the runtime environment in the software field. Playing with the word "environment" in a system development context,

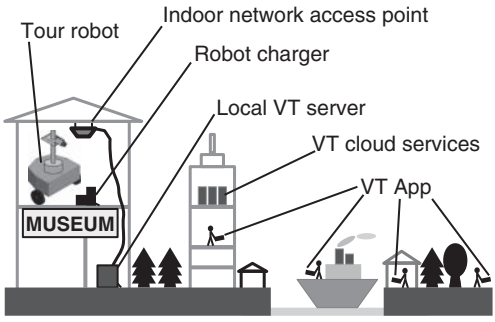


Figure 11.14 An architecture view from the system deployment perspective in informal representation, for the example in which the virtual tour is offered through a museum.

however, allows us to see that there is a notion of a different kind of environment attached to the *system deployment perspective* like it is exemplified in Figure 11.14: The figure shows that the robot is used in an indoor environment, such that environmental conditions like rain, wind, or fog do not need to be taken into account when designing it.

And finally, also location information can be addressed from system deployment perspective according to Figure 11.14: It is clear that the distance between the installation in the museum and the rest of the system can be considerably large. One can deduce that response times in communication between the "VT App" of the virtual tour client and the robot may become an issue. In the context of software architecting, Rozanski and Woods [213] define a "location perspective" to which they attach these kind of distance effects that result from the absolute location of system elements. In our case, we see no necessity to define this additional perspective, because our generalization of the deployment aspect from software architecting into a system deployment perspective for systems architecting produced location information available within the system deployment perspective.

The notion of perspectives has allowed us to handle the different kinds of information to be provided by the model. From the system deployment perspective, one can deal with location data as an additional piece of information about elements whose home is in other perspectives. For example, the system elements whose location is shown in Figure 11.14 belong to the physical perspective. But it is also possible to make a system deployment perspective with functional elements or layers or even combinations. For example, one can use the system deployment perspective to assess which functions of the Virtual Tour system will be deployed to system elements that are always present at the facility itself. This will allow to consider, e.g., whether a service technician needs to bring a mobile device with an app to service robots in the facility in which they are used. As a consequence, one can determine whether the user interface layer needs to be deployed to the local VT server on site or only to the mobile device of the service technician.

The system deployment perspective can also facilitate interface specifications. In the case of the Virtual Tour system, the architect may like to know if the power connection of the indoor network access point should be at the top or at the side of the device. Via the appropriate view of the system deployment perspective, one can answer this question. In the example in Figure 11.14, the architect would conclude that the power connection is best foreseen at the top of the indoor network access point. Liang and Paredis [158] point out that a position property is a possible part of a port specification. In the example of the indoor network access point, such a property could be used to model the position of the power connection. It would become visible via a view of the system deployment perspective.

Last but not least, the system deployment perspective can be chosen for defining constraints on positions or distances. For example, there is a maximum

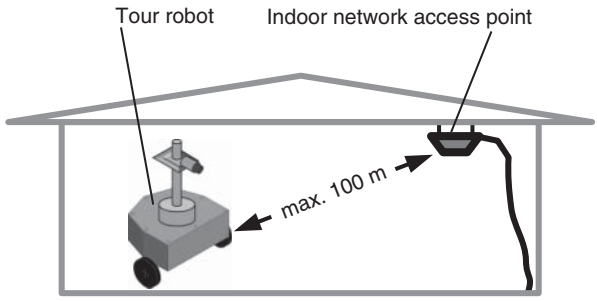


Figure 11.15 A constraint in the system deployment perspective in informal representation.

transmission range of network access points in a building. Figure 11.15 shows an example of expressing the corresponding constraint via the system deployment perspective.

In summary, the system deployment perspectives can provide architecture stakeholders with

- Location-related information
- Spatial and distance-related information
- Environment-related information that specifies in which locations certain system elements are exposed to the environmental effects identified during work on the system context (see Section 10.2.2) and which of the system elements are affected.

Spatial information of the system deployment perspective cannot yet be modeled in SysML very intuitively. Positions can, however, be modeled as properties of blocks and constraints on positions can be modeled as constraints in parametric diagrams. As an example, Figure 11.16 shows a SysML representation of the constraint from Figure 11.15 in a parametric diagram. It is expected that SysML v2 (see Appendix A.7) will offer more modeling support for the system deployment perspective.

11.7 Other Perspectives

Of course, this book cannot present an exhaustive set of perspectives. There are as many perspectives as there are kind of information to process about the system-of-interest. As a system architect, you may thus have to define your own perspectives in case the ones given in this book are not suitable.

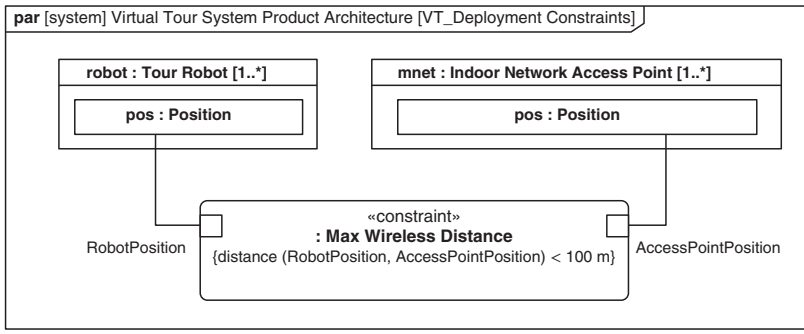

par [system] Virtual Tour System Product Architecture [VT_Deployment Constraints]
robot : Tour Robot [1..*]
pos : Position
mnet : Indoor Network Access Point [1..*]
pos : Position
«constraint»
: Max Wireless Distance
{distance (RobotPosition, AccessPointPosition) < 100 m}
RobotPosition
AccessPointPosition

Figure 11.16 The constraint from Figure 11.15 in the system deployment perspective in SysML representation.

There are two perspectives that are too much overlapping with the system deployment perspective to deserve an own section in this chapter, but sill they shall briefly be mentioned:

The *geometry perspective* is one from which geometry and shape are in focus. While the system deployment perspective specifies rough geometries and shapes of the overall system configuration, a geometry perspective could also be about specific geometries and shapes of subsystems. This is the information typically found in the repositories computer-aided design (CAD) software. Instead of integrating the geometry perspective into a SysML model, one could thus investigate possibilities to refer from, e.g., the elements of the product architecture to the corresponding elements in the repository of a CAD software. More SysML support for this perspective is expected from SysML v2 (see Appendix A.7).

In this context, it is worth mentioning some recent research that looked at the possibility of also interlinking model elements of the functional architecture with CAD data: The FAS4M project [96] was a research project that was funded by the German Federal Ministry for Economic Affairs and Energy. Its goal was to bridge the methodological gap between functional architectures and a structural specification of the system, including the idea of interlinking system models in model-based systems engineering with geometrical and shape data as today used in the area of CAD. The project lead to the discovery of one possible indirect way of interlinking, using certain sketches of mechanical shapes to bridge the functional and geometry perspectives. The project furthermore discussed various aspects of interlinking functional models with structural CAD data. A detailed summary of the project's findings would be beyond the scope of this book. The interested reader can refer to the project's publications, which are partially in German language (e.g. [96, 174, 175]), or directly look at some derived publications in English language,

which have particular focus on the mentioned approach of using sketches (e.g. [95, 173]).

The *topology perspective* is one that is frequently encountered in the communication networks domain and several other areas. It looks at the structures of interconnections between system elements. As opposed to the system deployment perspective, it will not focus on distances and precise locations. Instead of showing the distribution of tour robots and apps across the world, the system topology perspective would, e.g., be concerned with the question how many different communication routes there are between the user's handheld application and the tour robot.

11.8 Relation to the System Context

11.8.1 Validity of the System Boundary

Modeling done from any of the perspectives mentioned in this chapter usually needs to be compliant with the system boundary, which limits the scope of what will be considered to be part of the system-of-interest. This scope limitation is usually made explicit when systems engineers define the system boundary via the system context (see Section 10.2.2). Based on this definition, one can ensure that model elements will represent what resides inside the system boundary or in the system context, but nothing more. This limitation of the scope shall remain the same for a given system, no matter which perspective has been chosen. Even if we recommend to create models in compliance with such scope limitations, one needs to be aware that it is not always possible to apply the same rigor to informal illustrations. An example is given by some of the informal figures in this chapter, in which trees or building shapes were shown to provide an informal sense of location.

Right on the system boundary, there are interaction points for context interactions. These context interaction points can be in scope of multiple perspectives, because they are "ports" that can have multiple aspects [158], which often cannot be addressed by one perspective alone. In general, the interaction points of one perspective are a subset of the interaction points of the complete system context, because not all aspects of all perspectives may be relevant for each of the interaction points. For example, it may not make sense to model the layer perspective for a mechanical joint to be connected to an external system.

Section 11.9 will address the mapping between different representations of the same interaction point between different perspectives.

11.8.2 Using the System Context as a Part of the Stakeholder-Specific Views

Showing the system's inner elements in combination with the system context can help clarify the system architecture and the system elements' interactions with

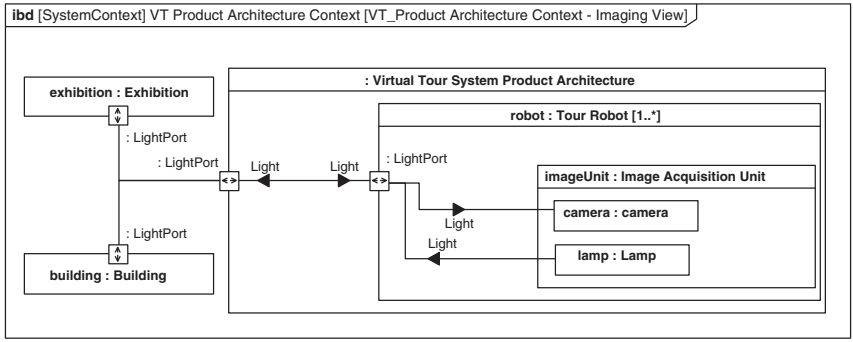


Figure 11.17 Example view on the Virtual Tour system, taking parts of the system context into account.

actors outside the system boundary. An example is shown in Figure 11.17. The example shows how light propagates from the robot to its environment and back for imaging purposes. In this example, SysML internal block diagram is used to show the inside of a system context object.

Let us also remember that we used a joint representation of the system context and its inner elements from the layered perspective, in order to show the role of the different layers in the system's interaction with external actors (Figure 11.13).

11.8.3 Special System Context View for Verification

Verification people are architecture stakeholders with regards to the system architecture. Their concern is the proper verification of the system. Even though certain verification activities should help ensuring that the system works in its utilization stage, the verification may use the system different from its typical utilization. For example, humans pressing a button may be replaced by machines that can cause well-defined stress conditions through high repetition rates of button presses. Human-machine interfaces may be used by automata that can perform different tests in a fraction of the time a human user would need for the same activity.

The different nature of actors in the system's verification environment can lead to a system context that is different from the normal *operational context* (see Section 10.2.2). This can be represented by means of a specialized system context specification, which describes the *verification context*. The verification context may allow for multiple verification methods. However, if the only method foreseen in the given setup is *test*, then the verification context can also be called the *test context* [150].

In the case of the Virtual Tour System, one could think of a highly automated test environment, in which external influences like user commands to the handheld application or visual input to the robot's camera are emulated by

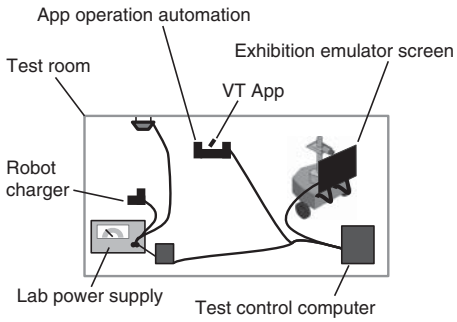


Figure 11.18 An informal representation of the verification context.

automated equipment (see Figure 11.18).[3] For the sake of measuring power consumption of certain subsystems, a laboratory power supply could be used instead of a facility's electrical installation – and all equipment could be controlled and monitored by means of a test control computer.

The special verification context according to Figure 11.18 can be modeled in SysML in the same way as the operational context. The corresponding modeling approach was presented in Section 10.2.2. The system context is typically shown via an internal block diagram that looks into a block with the stereotype «SystemContext». This block has one part property representing the system and one for each actor. For modeling the verification context, the «SystemContext» block simply needs to be a different one, whereas the «system» block would be the same one as in the operational system context. This way, a different context interaction of the system can be modeled. An example of a resulting verification context specification in SysML is shown in Figure 11.19.

11.9 Mapping Different System Elements Across Different Levels

Holistic understanding of the system often triggers the need to show how entities from different specification levels are mapped to each other. Different perspectives enable the description of the system via different kinds of information, but in the end, there needs to be a consistent truth that is followed by the system under development. This section describes how to map the entities to each other in order to make the resulting architecture description an interconnected holistic description

3 Of course the initial cost for establishing such elaborate infrastructure only pays back with high market volumes or high unit cost. A company only making virtual tour systems might not be able to afford it, but maybe a company that is world market leader in all kinds of robotic applications would already have such equipment in place and would also use it in cases it liked to develop and verify a virtual tour system.

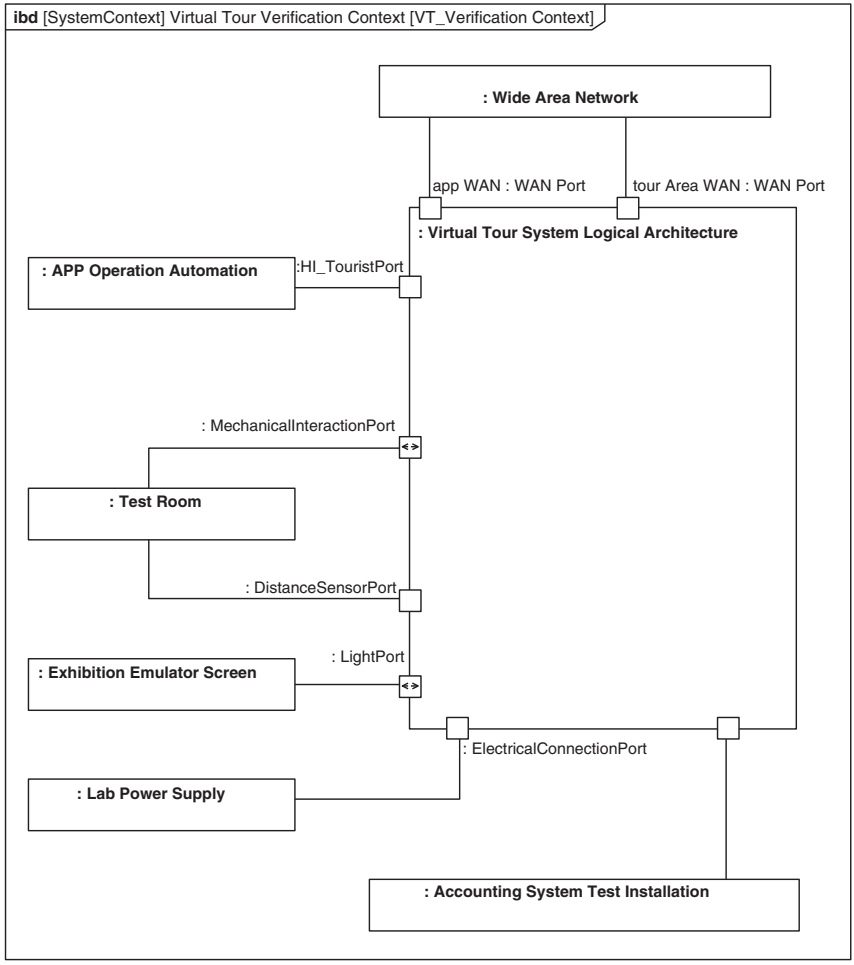


Figure 11.19 The verification context specification via a system context diagram in SysML.

of the system as a whole. The resulting mapping leads to a perspective by itself. It overarches the perspectives from which we can see the mapped entities – but it also represents a new kind of information, the "mapping" kind of information.

11.9.1 Functional-to-Physical Perspective Mapping

The mapping of functional blocks to physical blocks is called functional-to-physical mapping [103]. Ulrich [253] distinguishes different types of architecture, dependent on the kind of mapping: If each functional block is mapped to one

Table 11.1 Modular vs. integral Architectures according to Ulrich [253]

Functional blocks to physical blocks relationship	1:1	$1 : N$	$N : 1$	$M : N$
Type of the architecture	Modular	Integral	Integral	Integral

Source: Based on [253].

physical block, he talks of *modular architecture*, whereas an approach of providing functions via a combination of different physical blocks is called *integral architecture* (Table 11.1). Different kinds of nonfunctional requirements may lead to different kinds of architecture. For example, integral architectures may be a suitable approach for microelectronics, because microchips can be made smaller if the same processing unit is used for contributing to different functions of the system-of-interest.

The mapping can be expressed by means of a table (allocation matrix) or it can be modeled. In SysML, functional-to-physical allocations are defined by means of «allocate» relationships that are drawn between the part properties in the functional architecture and those in the physical architecture.

Table 11.2 shows an allocation matrix of the Virtual Tour system for mapping elements of the functional architecture to elements of the product architecture. Note that typical SysML modeling tools can also create the table representation according to Table 11.2. A convenience feature of at least some of the SysML tools is to automatically create «allocate» relationships based on a double-click in the corresponding cell of an allocation matrix. Section 17.7 will show examples of a functional-to-physical mapping in SysML, this time for the mapping of elements from the functional architecture to elements of the logical architecture, and hence with a similar, but slightly different content than the table shown here.

In practice, the pure allocation may not be enough to clarify the contribution of the different physical blocks to providing a function. This is particularly important to consider in the area of integral architectures. A more detailed system design may be needed to specify the way of mapping functions to physical blocks. One way of achieving this is to create relationships between functional blocks, physical blocks, and detailed requirements toward the physical block, specifying how the block has to contribute to providing a function. In order to do this, elements from different perspectives of the system architecture description have to be repeated and the relationships between requirements, architecture elements and partitioning decisions have to be explained. An example is shown in Figure 11.20. The figure also shows how to address nonfunctional requirements, which are at risk to be forgotten in a very functional-oriented approach. The nonfunctional requirements for various subsystems are derived from the system-level nonfunctional

Table 11.2 A functional-to-physical allocation matrix between the functional architecture and the product architecture for some sample elements of the Virtual Tour system

	Customer management	**Robot management**	**Tourbot Functional Architecture**	**Tour Management**	**I/O Virtual Tour Customer**	**I/O Tour Guide**	**I/O Tour Planner**
VT Cloud Services	×	×		×			
VT App					×	×	×
Local VT Server	×	×				×	×
Local VT Server Application	×	×				×	×
Indoor Network Access Point		×		×			
Robot			×				
Robot Charging Station		×	×				

requirement. This is indicated in the model by means of the extended derive relationship «exDeriveReqt» that has been introduced in Section 9.1.

Figure 11.20 shows very detailed considerations that deviate from the kind information shown in the typical pieces of architecture description that were shown in this book so far. The deviation is not only due to the increased level of detail, but also related to the fact that here, considerations and rationales are made for deriving a lower level of requirements specifications – or in other words: for bringing the expressed idea close to something that can be implemented. In such cases, we may consider not to talk about system architecture anymore, but about system design (see Section 5.6.2). When applying this approach to Figure 11.20, then the system design will be given by the part properties of the functional and physical perspective, the rationale element, and the «allocate» and «exDeriveReqt» relationships. Since the system design explains a refinement of architecture elements to an implementable level of detail, it is often not possible to explain it without the repetition of other model elements (like, e.g., requirements and system elements in the example shown in Figure 11.20). The example in Figure 11.20 also

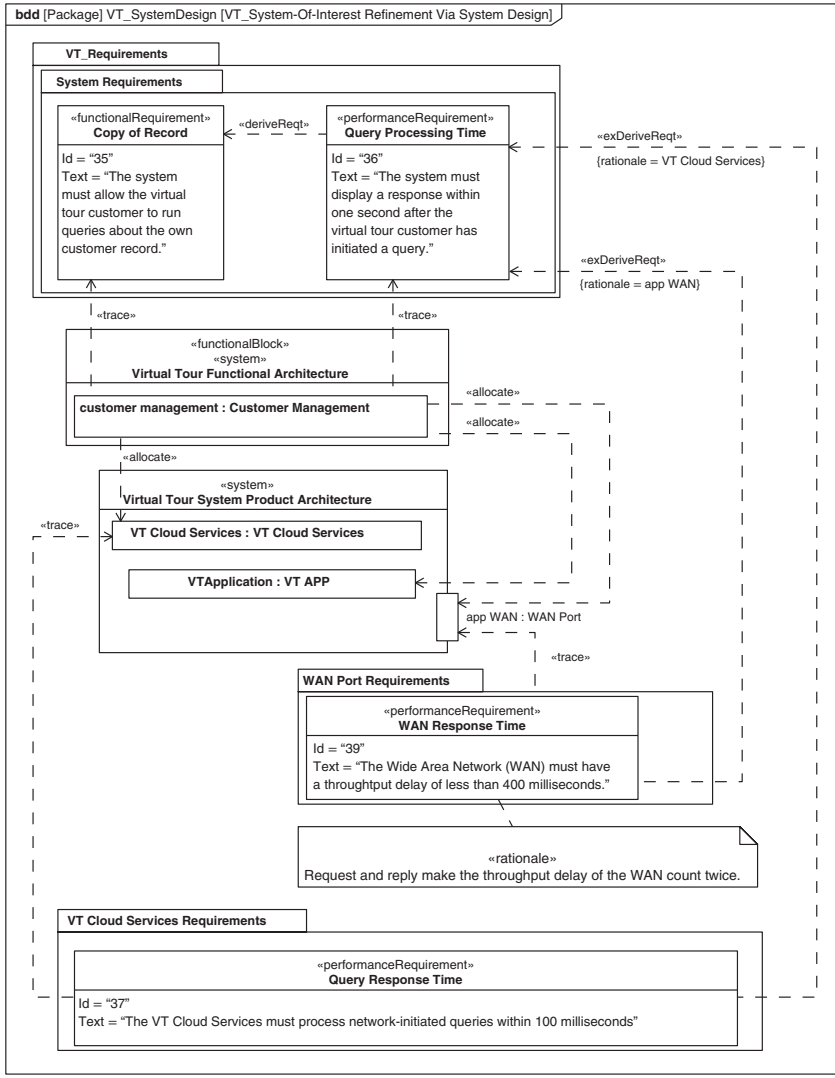

bdd [Package] VT_SystemDesign [VT_System-Of-Interest Refinement Via System Design]
VT_Requirements
System Requirements
«functionalRequirement»
Copy of Record
Id = "35"
Text = "The system must allow the virtual tour customer to run queries about the own customer record."
«deriveReqt»
«performanceRequirement»
Query Processing Time
Id = "36"
Text = "The system must display a response within one second after the virtual tour customer has initiated a query."
«exDeriveReqt»
{rationale = VT Cloud Services}
«exDeriveReqt»
{rationale = app WAN}
«trace»
«trace»
«functionalBlock»
«system»
Virtual Tour Functional Architecture
customer management : Customer Management
«allocate»
«allocate»
«allocate»
«system»
Virtual Tour System Product Architecture
«trace»
VT Cloud Services : VT Cloud Services
VTApplication : VT APP
app WAN : WAN Port
«trace»
WAN Port Requirements
«performanceRequirement»
WAN Response Time
Id = "39"
Text = "The Wide Area Network (WAN) must have a throughtput delay of less than 400 milliseconds."
«rationale»
Request and reply make the throughput delay of the WAN count twice.
VT Cloud Services Requirements
«performanceRequirement»
Query Response Time
Id = "37"
Text = "The VT Cloud Services must process network-initiated queries within 100 milliseconds"

Figure 11.20 System design and its interconnection and overlap with system elements and requirements.

explains that the system design of a system-of-interest may provide a rationale for certain requirements toward one or more system elements of this system. The system design activity, as assumed here, would fall under the design definition process according to ISO/IEC/IEEE 15288:2015 [115]. Since this book is dedicated to the architecture definition process, it will not elaborate further on the system

design – even though in practice, the system design may fall into the area of responsibility of a system architect.

11.9.2 Mapping More Perspectives

As we have seen, the deployment perspective is based on system elements from another perspective. By using these system elements, a relation to the other perspective is automatically given.

From the behavioral perspective, we usually see the behavior of one or more system elements. Again, the relation to these elements is implicitly in their use within the views of the behavioral perspective.

The functional-to-physical mapping has already been described. What remains is to describe mappings in which functional, physical, and layered perspectives exist. The following has been based on Dänzer et al. [61]:

- «allocate» relationships can be used to link the part properties that result from functional blocks with the ones resulting from «layer» blocks, where the direction should be the same as in functional-to-physical mapping, when looked at from the perspective of the functional block. It is recommended to split functional blocks in such way that the resulting blocks can each be linked to one unique layer. Functional blocks that have been created only for the corresponding reason of having a unique mapping to layers, and not based on domain knowledge, should be marked. To do so, we can assign the stereotype «layered» and concatenate the block name with the layer name. In the case of the "Robot Management" function running on the VT server's application and technical service layer, we could, e.g., decompose the functional block "Robot Management" into a block "Robot Management – Application Layer" and "Robot Management – Technical Service Layer."
- Finally, part properties that are specializations of «layer» blocks can be linked with physical blocks by means of an «allocate» relationship, where the direction is the same as in the mapping from a functional block, seen from the perspective of the physical block.

11.9.3 Mapping Different Levels

In this book, the functional, perspective and the logical and product level of physical architecture are described. In case all of them exist, a logical-to-product mapping can be made by means of SysML «allocate» relationships, like shown in Figure 11.21.

In certain cases, it may be beneficial to derive the product architecture from the logical architecture by means of a specialization relationship. In that case, the part

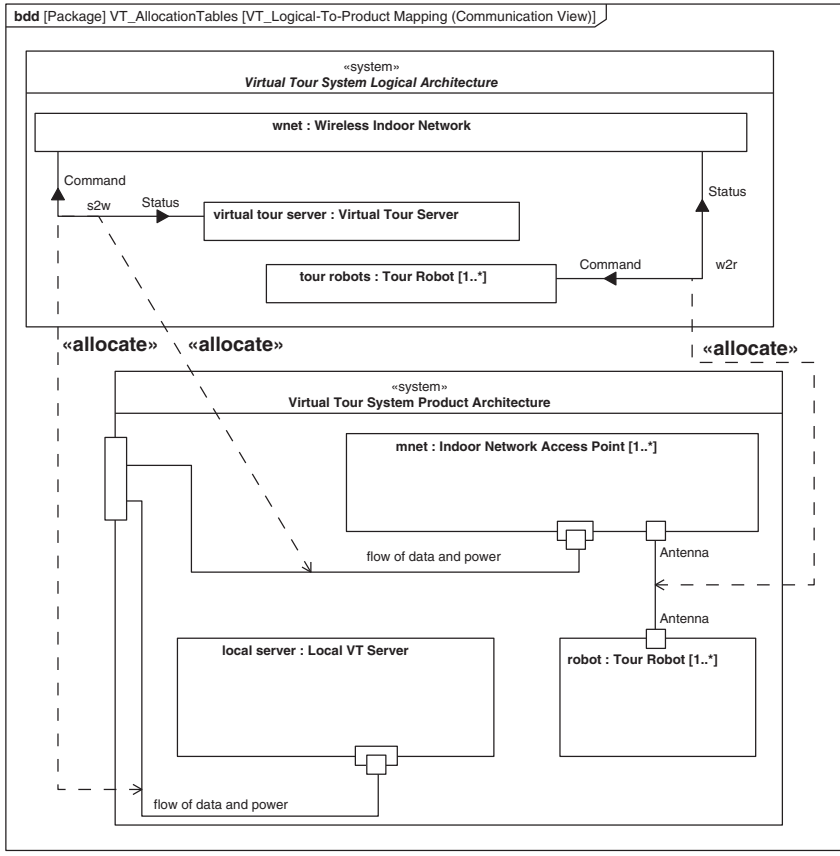


Figure 11.21 Logical-to-product mapping in a communication-focused view.

properties of the logical architecture will be inherited into the product architecture. Then, those part properties can be redefined for more concrete definition in the product architecture. This is particularly beneficial if parts of the logical architecture are so technically concrete that a 1 : 1 reuse into the product architecture is possible.

As always, there are some pros and cons to this approach:

- Pro specialization instead of «allocate»:
  - There is a tight connection between the logical architecture and the product architecture. The model ensures that the product architecture remains compliant with the corner stones that are set by the logical architecture.
  - In case of overlap between the logical level and the product level, reuse of modeling is possible from one level to the other.

  - The «allocate» relationship establishes a loose connection between the logical architecture and the product architecture, resulting in the potential for inconsistencies between both.
- Contra specialization instead of «allocate»:
  - The tight connection between logical architecture and product architecture makes changes in the logical architecture directly influence the product architecture.
  - Redefining elements of the logical architecture may occur so often that the overview is lost.
  - When using an «allocate» relationship, then the development of the logical architecture is decoupled from the one of the product architecture.

11.10 Traceability

A special view is the traceability view. One typical traceability view ensures that the concern of finding the requirement behind an architectural solution can be addressed. The modeling of this kind of traceability has been discussed in Section 9.1.

11.11 Perspectives and Architecture Views in Model-based Systems Architecting

11.11.1 Creating Different Architecture Views in a Model-Based Approach

A text document is sufficient to establish different perspectives and different views (document-based approach). For example, the functions in Figure 11.1 can be represented as a bullet list, preserving both the functional focus of the functional perspective and the hierarchical representation of the given view:

- System
  - Customer Management
  - Robot Management
  - Tour Management
  - Tourbot Functional Architecture
  - I/O
    * Virtual Tour Customer
    * Tour Guide
    * Tour Planner

If text is not sufficient, free-style diagrams like Figure 11.1 can be used to complete the architecture description.

A model-based approach is thus not necessary in order to produce sufficient architecture documentation. Conversely, this means that model-based systems architecting will have to compete with intuitive-to-read documentation that can be created by means of free-style writing and drawing. The system architect should thus strive for generating views that are as close to the stakeholders' preferred representation as possible. In the case of Figure 11.1, it is obvious that the native SysML representation according to Figure 11.2 will most likely satisfy the architecture stakeholders equally well. A good modeling tool can create figures like Figure 11.2 automatically, once the displayed information has been entered in the model. Some tools even offer automatic layout of diagrams.

In case the architecture stakeholders' preferred representation cannot be produced within the possibilities of the modeling language or the used tool, reporting and document generation can be considered (see also: Section 8.3). The more specialized the views need to be the more difficult or costly it will be to generate them from a model. In order to justify the resulting cost of model-based systems architecting, one should thus be aware of its benefits.

On the benefit side, the model-based approach offers means of ensuring consistency between different perspectives and architecture views in an effective manner. For example, if "Customer Management" needed to be renamed to "User Management" in the above example, a model-based approach would only require one renaming action in the model, and all different views would automatically use the term "User Management" from the moment of the first renaming onward. Model-based systems architecting thus increases efficiency and consistency in documenting system architecture with different perspectives that serve different purposes and different architecture views that address different stakeholders' concerns.

In balancing cost against benefits, the system architect should thus find the appropriate trade-off between serving the architecture stakeholders with their preferred representation of information and keeping the needed infrastructure (modeling tools, reporting tools, etc.) affordable and maintainable. In case of doubt, we recommend to keep it simple, this means to try avoiding complex infrastructure. In our experience, stakeholders are more reluctant to work with information from a complex IT system than to accept a representation of data that is not 100% according to their preference. This may be due to the fact that complex IT infrastructure has less availability[4] than simple tools, and availability of data is an important success factor in business.

---

4 If one tool has a probability of unavailability of p, then a tool chain of N tools of equal availability have the probability of $N \cdot p$ under some typical assumptions.

We often heard architecture stakeholders, but also architects justify free-style architecture documentation by pretending that the consistency can be easily maintained without a modeling tool. However, we also saw how much confusion can be created if inconsistent names are being used for elements in the system – or even worse: If the same term is used for two different things, e.g., by naming a subassembly exactly like one of its parts. When such confusion is discovered, it has often spread out across the architecture stakeholders, in large organizations even throughout different countries and documentation repositories. The cost of correcting such problematic use of terms can be moderate to high, if just the effort for the correction of documents is measured. However, the real damage caused by inconsistent terminology can only be assessed if more soft factors like conflicts arising from misunderstandings and architecture stakeholders' effort for learning corrected terminology are taken into account.

We therefore recommend the model-based approach together with the notion that the model is the single source of truth. Team exercises like the joint creation of a domain knowledge model according to Section 10.2.5 in a workshop can help the stakeholders understand how ambiguous their use of terms is if not consolidated in one model.

11.11.2 Using SysML for Working with Different Perspectives and Architecture Views

The different diagrams in the previous sections show examples of representing different views in both informal notations and in SysML. Some concrete hints for organizing models with different perspectives in SysML are given in Section 9.9. Here, we discuss how SysML can address the different perspectives and how to provide suitable architecture views for different architecture stakeholders.

It can be seen that SysML can represent tree-like breakdown structures very well (e.g. in Figure 11.2). It can also represent allocation matrices where the mapping of different perspectives is described.

What engineers would often describe in their jargon as "block diagrams" can be provided by the SysML internal block diagrams (e.g. Figures 11.4 and 11.6). Interfaces can be shown on a high abstraction level by means of internal block diagrams and then refined, either by specifying details of data flows via a domain knowledge model (Figure 11.5) or by adding details about ports to the specification of their port types (Figure 11.11). We highly encourage system architects to use these SysML notations for interfaces definitions during workshops with the corresponding architecture stakeholders, because a system architect should have a commonly understood and agreed specification of interfaces as one key objective (Section 13.2.1). SysML helps creating well-understood interface specifications by being both precise and intuitive.

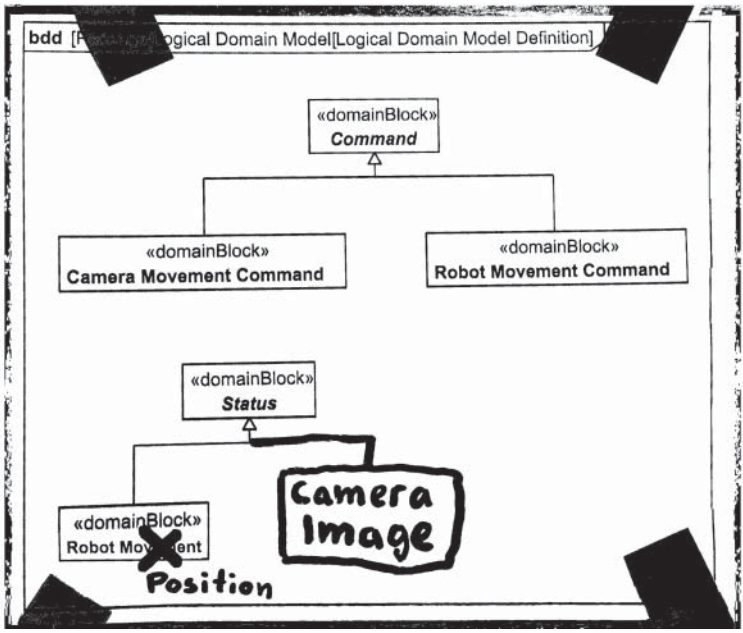


Figure 11.22 The partially hand-made version of Figure 11.5, as it could have come out of a workshop with stakeholders of the tour robot's higher communication layers.

In case the architecture stakeholders are not familiar with SysML, particularly the approach via a domain knowledge model is an intuitive one that can be used after a short explanation of the involved SysML syntax. Only few SysML users are so skilled with a modeling tool that they can use it live in a workshop with stakeholders. If a version of such a diagram already exists prior to the workshop, then we usually print it on a poster and bring it to the workshop, allowing the stakeholders to draw on it or put sticky notes onto it. Figure 11.22 shows how Figure 11.5 would have looked like if it had been worked on during such a workshop.

In many cases, we would recommend using flipcharts and markers (so-called "PAPS tools" [5]) for creating domain knowledge models like the one in Figure 11.5. For all those who think that flipchart are suitable modeling tools for parts of their daily modeling use cases, we recommend the book "Agile Modeling" by Ambler [18].

5 "PAPS" means "Pen And Paper System." The acronym is quoted from Holt and Perry [104], who discuss the role of PAPSs for the professional systems engineer.

11.11.3 The Importance of Architecture Viewpoints in Model-Based Systems Architecting

While document-based architecture descriptions may be derived from a document template with instructions to insert certain views of the system, the model-based approach requires a description how to generate the needed architecture views from the model. This description can be derived from a proper definition of the related architecture viewpoints. In a more sophisticated vision by Gerritsen et al. [89], the viewpoint definition can be made even formal enough to allow views to be generated and updated automatically. Based on the quoted work [89], one can think of a model containing architecture viewpoint definitions, but no architecture views. The architecture views are then created on demand, using the viewpoint definitions. This vision usually only works if very strict modeling conventions are defined upfront and followed by each modeler, because only then the model is structured enough for allowing a computer program to find and interpret information and to generate the intended architecture views. Even though the outlined vision sounds promising, we have not seen extensive use of it in practice. The notion of not storing views at all is very radical and comes with a practical problem related to configuration management: How to baseline a set of views if they are not stored at all?

A maybe interesting compromise between the automatic on-demand creation of views based on a formal viewpoint definition and the storage of views has been taken in the so-called AID plugin for modeling tools [20], which has already been briefly discussed in Section 8.3. The plugin can automatically create certain SysML diagrams, which are very similar to the ones shown in Section 11.2.3 or 11.3.2, and it can automatically color blocks to distinguish them from other blocks on the diagram – very similar to the black fill color usage we showed in Figure 11.3. At the time of writing this book, the plugin uses very few hard-coded viewpoint definitions and therefore only supports a very narrow set of views. Once a SysML diagram has been automatically created to present a view, the diagram can be stored and baselined. After model updates, the AID plugin can check if diagrams are up-to-date or even update them automatically, according to the viewpoint definition.

Let us assume that the described plugin is a representative example of current automation possibilities. Let us also once more recall the plugin's just mentioned limitations. Let us furthermore revisit the more general statements about currently available automation possibilities in Section 8.3. Then, the conclusion may be that

approaches for automatic generation and updates of architecture views are not yet broadly available and still evolving. To the authors' knowledge, the corresponding solutions are not yet available off-the-shelf to an extent that, e.g., the SysML diagrams as shown in this chapter could all be created automatically, just based on a viewpoint definition.

12

Typical Architecture Stakeholders

12.1 Overview

Stakeholders in the context of system architecture are the main topic of this chapter. However, we first need to set a larger context, because stakeholders are a topic in requirements engineering (RE) as well.

Requirements toward the system-of-interest lead to concerns in system architecture. Requirements engineers capture requirements in direct dialogue with people or entities they call "stakeholders" or "stakeholder representatives" in a requirements engineering context. Users of the system are very important stakeholders in requirements engineering. Furthermore, acquirers, regulatory bodies and entities performing marketing, production, training, operations and maintenance will typically have requirements toward the system-of-interest. It is a matter of taste whether these people or entities are explicitly identified as stakeholders in a system architecture context or whether the system architect will consider the requirements engineering people to be the actual architecture stakeholders when the concern is to satisfy requirements.

Apart from their role as potential providers of requirements, some people or entities may have more interaction with the system architect and more concerns to be addressed during systems architecting (SA) than covered within the scope of the requirements engineering processes. They are the ones who should collaborate with the system architect, which makes them potentially important stakeholders in the context of system architecture, no matter whether they have requirements toward the system-of-interest. We call these kinds of stakeholders the architecture stakeholders. They include people who are no stakeholders in the context of requirements engineering but have a stake in the system architecture.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

The further sections in this chapter are about the architecture stakeholders. Selected aspects of the system architect's collaboration with them will be discussed. Each of the following sections is dedicated to one usual architecture stakeholder inside a typical organization and the organizational interface to the system architect. Depending on the nature of the business and the involved organizations, the described entities may exist as distinct departments or as persons with certain roles or tasks. They may exist in the organization in which the system architect works or in one of its business partners. The selected stakeholders and topics are not generally applicable, because they are examples. You may consider using this chapter as an inspiration when analyzing who are the architecture stakeholders of your own system-of-interest.

We describe the collaboration between the system architects and the selected architecture stakeholders as a win-win situation, because the architecture stakeholders may be looking at their own wins when asked to reserve time for cooperation with the system architect. The tables belonging to each of the following sections characterize the cooperation between the system architect and the respective architecture stakeholder in exemplifying what each of the parties has to give and what can be gained in return.

This chapter covers stakeholders that are often encountered for typical systems-of-interest. It should be noted though that such a system is typically surrounded by *enabling systems* (refer to ISO 24748-1:2018 [116]), which are used, e.g. during development, production, support, and retirement of the system. The enabling systems have to be compatible with the system-of-interest and have interfaces with it, i.e. the external interfaces of the system. These interfaces will typically be found during the work on the system context (Section 10.2.2). The people concerned with the enabling systems may be additional architecture stakeholders for the system-of-interest. They are not comprehensively discussed in this chapter, because who they are and how they interact with the system architect is strongly dependent on the system-of-interest and the organizations involved in its life cycle.

12.2 Requirements Engineering

The role of requirements engineering as an architecture stakeholder has been briefly discussed in Section 12.1. Requirements engineering people must communicate the requirements to be satisfied with a certain system architecture to the responsible system architecture people. "Communicate" means ensuring that mutual understanding is created by means of at least partly verbal direct and bi-directional communication.

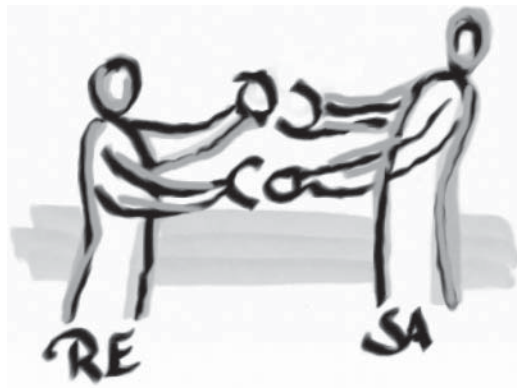


Figure 12.1 Collaboration between requirements engineering (RE) and systems architecting (SA).

Systems architecting results in system architecture that satisfies the requirements. To check whether the requirements have been correctly understood by the individuals involved in systems architecting, the system architect should communicate with the requirements engineer. In this case, the communication ensures that an input to system architecture has correctly been received.

If requirements on a lower abstraction level need to be derived from the system architecture, there is also an output from the systems architecting discipline to the requirements engineering discipline. There is thus a bi-directional communication link between requirements engineering and systems architecting roles in cases with multiple abstraction levels. It directly corresponds to the relationships between requirements and architectures in the SYSMOD zigzag pattern we saw in Section 9.1. Figure 12.1 illustrates the bi-directional nature of the interface between requirements engineering and systems architecting: The upper handshake in the picture means that requirements engineering provides requirements as an input to systems architecting. The lower handshake means that systems architecting provides an architecture description as the basis for deriving requirements on a lower abstraction level. The picture has been provided by Tim Weilkiens [268] who made it for blogging about the need for close cooperation between requirements and architecture people and about the two directions of communications that are implied by the SYSMOD zigzag pattern.

Table 12.1 describes the cooperation between system architects and requirements people as a win-win situation.

12.3 Verification

Verification is supposed to show that the system has been created according to its specification. According to the V-model,[1] e.g. in a version according to Emes

1 A description of the V-model can be found in the Appendix B.

Table 12.1 Close collaboration between system architects and requirements people as a win-win situation

What requirements engineering people give	• Explanation of the requirements
What they get in return	• The requirements are being taken in for further processing • The system architecture is explained to them for the work on derived requirements based on the system architecture (see zigzag pattern)
Obligations toward system architects	• Requirements people hand over requirements to system architects and ensure that a common understanding is reached
What they can expect from system architects	• System architects account for the requirements in systems architecting • System architects give feedback if requirements are not clear or deficient • System architects ensure that system architecture descriptions are traceable to requirements • System architects provide architecture descriptions that can be used for deriving requirements on a lower abstraction level

et al. [70], *system tests* verify the fulfillment of system requirements, and *integration tests* verify the fulfillment of architectural specifications. Tests can be replaced by other verification approaches like inspection or demonstration. In any case, the expected nature of the system is derived from its specification and compared with the observed nature from verification. This activity requires several inputs that are either work results of systems architecting or can be derived from these work results:

- For assessing the fulfillment of architectural specifications, the people responsible for verification need the system architecture description. The system architect needs to communicate with them in order to ensure common understanding.
- The system architect can support the appropriate design or assignment of priorities in verification. For example, if the risk of failure in a certain state of the system-of-interest needs to be minimized, the system architect can provide views showing how the system can enter the given state. On this basis, test cases provoking the system to enter the given state can be designed or prioritized.

- Methods based on so-called equivalence classes aim at coverage of system requirements with tests. They avoid the typically unpractical or even impossible attempt to cover the full input vector space of the system with test vectors. Instead, they recommend the use of test vectors that cover a set of "equivalence classes" of the input vector space with one test vector per equivalence class. The work by Richardson and Clarke [207] and the category partition method [195] in software testing have enabled system-level methods like classification trees [48]. Richardson and Clarke [207] propose looking at both the requirements and the solution when designing tests for coverage. This indicates that the design of system tests may require knowledge about the solution chosen for the realization of the system of interest, i.e. its system architecture. Practically, this means that the people who plan or design system tests are architecture stakeholders and that the system architect should closely cooperate with them.
- After changes in the system, regressive verification needs to assess that the changes produced the desired effect and did not produce side effects in the system. Only by having a thorough understanding of the system-of-interest and its inner connections and dependencies can one assess which steps need to be repeated during regressive verification. In other words, the knowledge of the system architecture is required in the design and planning of regressive verification. An easy way to make the required architecture knowledge available during the verification planning process is to involve system architects with knowledge about the appropriate parts of the system architecture in that process.

Another aspect of system architecture involvement in verification activities is the post-processing of failed verification. Particularly in system testing, the cause for a failed test step can potentially reside in multiple parts of the system (if not in the test procedure itself). Based on the system architect's expertise about the system structure, it is easier to narrow down the number of hypotheses about the potential cause of a failure.

People with responsibility in verification are thus architecture stakeholders and should have a strong interface with system architects.

Furthermore, the system has to be architected for verification. This may mean to make the system easy to inspect for verification by inspection. This also means that systems architecting should account for the fact that a system needs to be tested during its life cycle. For example, testability requirements may lead to system functions like e.g. dataloggers or to additional interfaces like, e.g. test access points. In Section 11.8.3, we have seen that it is possible to show a variant of the system context diagram that describes the system in its test context. Test access points should be part of such a context diagram, and the diagram should explain what is

Table 12.2 Close collaboration between system architects and verification people as a win-win situation

What verification people give	• Trust in the system architect's expertise about the system • Expertise on the intended ways of verifying the system
What they get in return	• The system is designed for verification • Verification design and verification planning are based on information about the solution • Regression testing is based on the assessment of impacts of changes
Obligations toward system architects	• Verification people involve system architects in verification planning • Verification people involve system architects in the analysis of failed test steps
What they can expect from system architects	• System architects provide expertise about the system-of-interest

connected to them during the verification of the system. The detailed modeling of the connections with test access points allows for a precise interface specification.

Table 12.2 describes the cooperation between system architects and verification people as a win-win situation.

12.4 Configuration Management

This section has nothing or little to do with the configuration of variants according to Chapter 18. Configuration management in the context of this section ensures that versions are tracked across different deliverables in system realization. These deliverables are called *configuration items*. They can be documents, models, and implemented or assembled elements of the system. Configuration management defines *baselines*, that means sets of versions that are considered the state of the configuration items at a certain moment or decision gate. It also enables the tracking of compatibility between different versions of different configuration items [146].

The system architect can help identifying the different configuration items in the system, based on the system models, e.g. of the physical perspective. Models in model-based systems architecting can be used as the single source of truth for the name of the different configuration items. The system architecture description can

Table 12.3 Close collaboration between system architects and configuration managers as a win-win situation

What configuration managers give	• Creation of understanding for the applicable configuration management strategy • Overview of system configurations
What they get in return	• Versioned system architecture description deliverables into baselines • An overview of the configuration items in the system and their compatibility
Obligations toward system architects	• Configuration managers explain the needs of configuration management • Configuration managers plan baselines together with system architects
What they can expect from system architects	• System architects deliver system architect's configuration items into baselines on time • System architects offer consulting services regarding the definition of configuration items and the assessment of compatibility

facilitate the assessment of compatibility between different versions of elements of the system, based on, e.g. interface specifications.

The system architecture description is a configuration item as well and needs to be versioned. In model-based systems architecting, one needs to decide whether the model itself or a set of views generated from it is entering a configuration baseline. In any case, views should be traceable to the version of the model from which they were created.

Table 12.3 describes the cooperation between system architects and configuration managers as a win-win situation.

12.5 Engineering and Information Technology Disciplines

The engineering or information technology disciplines produce work results that will in the end make the system follow the desired architecture. It is therefore necessary that these disciplines are committed to the system architecture in order to make it materialize. An architecture description has value only if it has been

created with stakeholders from the engineering or information technology disciplines that realize certain system elements.

In a close dialog between system architects and engineering or information technology disciplines, a common understanding about the system and its principles of operation is obtained. Ideally, a multidisciplinary dialog is established and maintained throughout the whole specification and implementation cycle. The system architect takes the lead in maintaining the dialog, mediating during trade-offs, and securing that work results are documented, validated, and made reproducible.

Typical representatives of engineering disciplines are:

- Division heads or group leaders
- Subsystem architects (e.g. software architects in the case of a software engineering disciplines)
- Lead developers or developers

The more hands-on knowledge the involved development stakeholders have, the more likely the resulting system architecture can be followed smoothly during implementation. The system architect should have an opportunity to talk to the developers who are allocated to the actual implementation work and not only to their leaders.

A particularly important aspect of the architect's work is to ensure that interface agreements are made, documented, and followed. This includes both interfaces between different subsystems and material, energy, or data flows between different functional elements within the same subsystem.

The negotiation of an interface agreement has to be made with the stakeholders on both sides of the interface. For example, an interface between the battery charger of the tour robot and the robot itself has to be negotiated with both the stakeholder representatives responsible for the battery charger and stakeholder representatives responsible for the robot.

It is important that engineering and information technology disciplines acknowledge the need to follow interface agreements or to help keep the system architect in control of the interface, e.g. by informing the system architect if adhering to the interface turns out to be impossible or discouraged. This seems to be trivial, but our practical experience has shown that stakeholder representatives who are for the first time collaborating with a system architect sometimes consider their job done when the first version of an interface agreement has been made. They might in consequence ignore the change processes that need to be followed when changing an interface definition. We have observed two patterns of violating interface agreements:

- People working on the element on one side of the interface change the interface definition without notifying the responsible for the element on the other side of

the interface, which leads to improper system performance on integrating the elements on both sides of the interface during system integration.
- People working on the elements on both sides of the interface change the interface definition together, but without involving the system architect, which leads to a discrepancy between the system architecture description and the architecture that is exhibited by the system and in consequence often to side effects that the people who changed the interface definition were unaware of.

We therefore recommend to system architects to periodically check whether interface agreements are still followed within the development organization, particularly if system architecture is new to the involved stakeholders.

It is important that system architects and stakeholder representatives from the engineering and information technology disciplines establish a mutual trust relationship. A challenge in that regard is the fact that the system architect has to interfere with the solution in the different disciplines in order to optimize the overall system performance. Stakeholder representatives from the different disciplines may consider this an act of getting involved in something that is not the system architect's business. It is necessary for the different domains to understand the need for the system architect to influence how the work in the disciplines is being done – but it is also necessary for the system architect to know where the own competence ends. Particularly, system architects who have been recruited from within the organization often have a background from one of the involved subject matters and have a particular difficulty not to get involved into the corresponding discipline's own business.

A successful introduction of systems architecting in an organization means that the mutual trust between system architects and subject matter domains is established and both sides understand that the boundary between system architecture work and subject matter work is unsharp and has to be sharpened on a case-by-case basis in the mutual dialog. It is a measure for the success of introducing systems architecting that the individual disciplines and system architects do not blame each other for exceeding their competence, but solve boundary questions together without any need for escalation.

We see three different scenarios how system architects and people from the engineering or information technology disciplines can collaborate:

- The system architect leads the definition of the system architecture
- The system architect has to run after architecture decisions that have already been made explicitly or implicitly by the concepts the individual disciplines have based their work on. The system architect has the license to document them and acts as system archaeologist.
- The system architect and the stakeholders from the different disciplines see themselves as a team that acts in a self-organizing way. Usually, multiple such

teams should be set up around different aspects of development, e.g. around a function or a set of functions of the system. Sometimes, such teams are known as feature teams, but then their scope usually comprises more than just the definition of the system architecture.

The above scenarios are not mutually exclusive, i.e. often the collaboration is a mixture between them.

Table 12.4 describes the cooperation between system architects and people in different engineering or information technology disciplines as a win-win situation.

In the particular situation of a collaboration between system architects and information technology or software engineering disciplines, it has to be considered that the former are ideally systems thinkers only, while the latter are typically computer scientists or people with similar background. A commonly encountered misunderstanding between these two specific kinds of collaboration partners in the context of model-based work is a different preference in the interpretation of model elements. For that sake, let's consider a model element (e.g. a SysML block) called "the system." There is a risk that the following conflicting interpretations will be made by default, if no further focus on mutual understanding is applied:

Table 12.4 Close collaboration between system architects and subject matter experts in the engineering or information technology disciplines as a win-win situation

What subject matter experts in the disciplines give	• Expertise about their domain • Willingness to change the design of a system element to make the whole better • Trust that the system architect will respect their own domain's competence
What they get in return	• Their stakeholder view toward interfaces with other subsystems will be respected by other stakeholders, e.g. engineers working on another subsystem • Their needs will be taken into account • They will be trusted that they are competent within their own field
Obligations toward system architects	• Subject matter experts report back if interface agreements cannot be met • Subject matter experts have system architects lead the change of interface definitions instead of changing interfaces alone
What they can expect from system architects	System architects keep subject matter experts updated about system architecture changes and involve them in systems architecting and change activities that affect them

- The information technologist or software engineer may assume by default: A model element "the system" represents a database entity or another memory object that can store or reference information about the whole system.
- The systems engineer may assume by default: A model element "the system" represents an object in the real world that is a materialization of the whole system.

The good system architect knows the above risk of confusion and actively mitigates it by explaining the different default interpretations and clarifying which one is up for debate. In the Virtual Tour system example, the system architect may draw a block (rectangle) on a white board and label it "Tour Robot." The system architect may then ask all stakeholders in the room whether the drawn rectangle represents something heavy and solid that can be touched or an entity in a database. This may help clarifying if all people in the room have the same understanding.

12.6 Project and Product Management

System architects can use their knowledge about the system-of-interest to help the project manager in achieving realistic plans and thorough project risk management. The system architect's knowledge about the system structure, interdependencies inside the system (Figure 12.2) and the complexity in different areas of the system are the key enablers in that regard.

Here is a list of typical tasks in a project to which a system architect should contribute in close cooperation with project management or the people to whom the tasks are delegated by project management:

- development and verification planning
- system integration planning
- technical feasibility analysis
- work breakdown
- analysis of technical dependencies between development activities (Figure 12.2)
- project risk management
- effort estimation

The project risk management activity is one that is closely related to the system architect's work. Trade-offs in system architecture often trade risk against effort. For example, the decision how generic an interface should be trades the high effort for making a generic interface against the risk of late changes in case an interfaces is initially designed for very few use cases only and needs to be extended upon discovery of a forgotten or newly required use case. The system architect does

Figure 12.2 © 2014 Jakob K., reproduced with permission.

not own the budget that tells how much effort can be spend and therefore needs to enter dialogue with the stakeholder who does own it. The project manager is often the stakeholder to involve in that regard. Our advice to the system architect is to not only report risks, but also proposals for mitigation strategies and the system architect's own preference among those. A more general discussion about risk management is due later, in Section 20.3.

In the context of effort estimation, the system architect also has to estimate the effort for the tasks that will need to be done within system architecture.

Of course, the workload for system architects depends on the nature of the project, but also on the concrete role description of the system architect in the given organization.

If the role description stays the same across multiple projects and across time, it is worth making a book keeping about the effort spent on different projects. This will improve estimates over time.

An example is shown in Figure 12.3: It shows an invented curve of the system architect's workload over time for systems architecting in a fictitious project and for collaboration with the stakeholders, like discussed throughout this chapter.

The curve in Figure 12.3 has two peaks:

- The first peak could relate to the system architect's activities that are involved in designing the system
- The second peak could relate to the system architect's activities in connection with verification (Section 12.3), but also to topics that occur during preparation

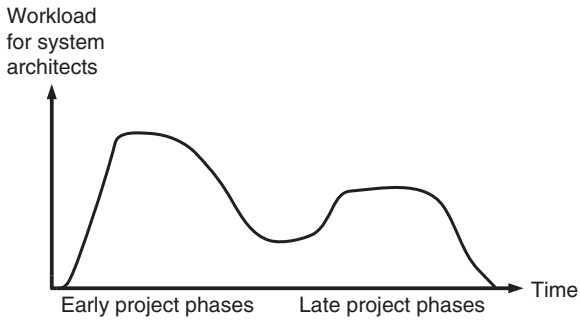


Figure 12.3 A fictitious example of predicting architecture workload over time.

of production and to change management activities that result from learnings after integrating close-to-final products toward the release.

Even though the curve in Figure 12.3 is fictitious, similar curves have been observed in real industrial projects.

Project leaders and system architects are two poles of a project: While project leaders have to drive the project forward and keep timelines, system architects have to ensure that technical work has enough technical depth and maintainability to ensure the required quality over the whole life cycle of the system architecture, not of the project. The project managers "pulls forward," the architect "pulls down" (Figure 12.4).

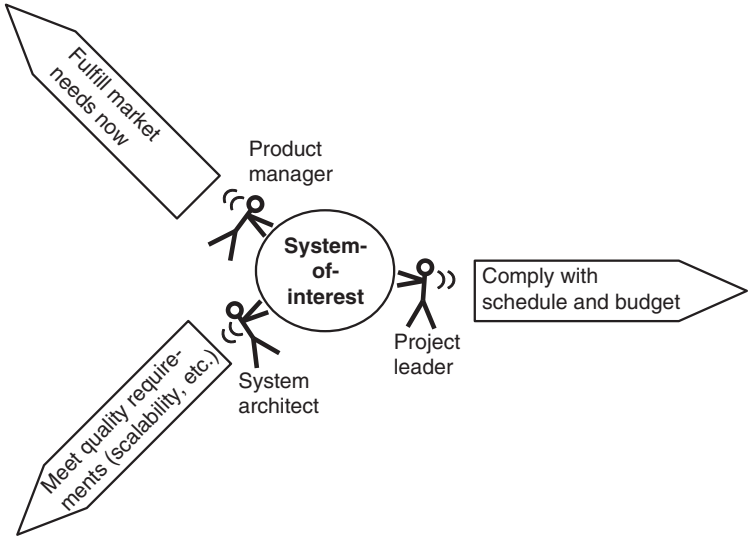


Figure 12.4 Split of concerns between system architect, project manager, and product manager.

One could deduce that a project driven by a project manager in close collaboration with a system architect would reach sufficient technical perfection within the right time, but there would be a risk that it would produce technology for its own sake ("happy engineering"). To ensure that market and customer needs are accounted for, someone should "pull up," i.e. ensure that the market needs are taken into account. This is a task of stakeholders like requirements engineering or product management. Typically, requirements management has captured post-processed stakeholder input, and the architect will get the stakeholder view from the requirements, or – better – from a requirements engineer who ensures that the requirements are understood and accounted for. Yet it may be necessary to bring the project leader, the system architect and product management closely together in order to run fast iterations between pulling up, down, and forward according to Figure 12.4 and thus bring a project on the right track. To keep the project moving into the right direction, they should stay in close contact during the whole duration of the project.

Table 12.5 describes the cooperation between system architects and project leaders as a win-win situation.

12.7 Risk Managers

Risk management also comprises the management of product risk. The people managing product risks also closely collaborate with system architects, because together they will develop the understanding of the product risks to be taken into account. There is no dedicated table of the related win-win situation, because the collaboration depends on the implementation of risk management in the organization.

12.8 Development Roadmap Planners

Development projects have a defined start point in time, an expected end point in time and an expected deliverable. The deliverable can for example be a concept, a product, a subsystem, or a new feature. Usually, several development projects can run in parallel. For larger organizations, there is typically such parallelism at any moment, whereas very small organizations may be in situations in which one project loads their full capacity. Even in the latter case, parallelism may occur during the transition phase in which one project ends and another one starts or if an in-market support case loads the development pipeline in parallel to a new development.

Table 12.5 Close collaboration between system architects and project leaders as a win-win situation

What project leaders give	• Time of their engineering resources for maintaining the multidisciplinary dialog between engineering and system architects • Trust in the return of investment that will be generated when investing into good systems architecting
What they get in return	• Overview and clarity about the project's deliveries • Improved communication in the project • Analysis of dependencies (Figure 12.2) • Predictability of cost and performance • Early information about risks • Better products (see Chapter 3)
Obligations toward system architects	• Involve system architects in planning in order to account for the correct dependencies (Figure 12.2) • Involve system architects in decisions that affect the system structure, system functions, or system behavior • Ensure that the time for proper systems architecting is allocated on the system architects schedule and on the one of the architecture stakeholders
What they can expect from system architects	• System architects offer consulting services during work breakdown, planning, technical feasibility analysis, and risk management • System architects provide overview documents and diagrams that help scoping the project • System immediately report if they perceive that plans are unrealistic or that the risk profile needs to be updated

There should be a list of both current and future development projects that provides overview about their assumed start dates, end dates, and deliverables. We call it the *development roadmap*. For simplicity, let people who are responsible for maintaining this list be called *roadmap planners*. In different organizations, this can be single persons or organizational entities.

System architects and roadmap planners have several good reasons to cooperate. Of course roadmap planners should also collaborate with other stakeholders in order to complete their puzzle. Here, we describe the puzzle piece that should be

addressed between the roadmap planners and the system architect. In our eyes, the most important aspects of this collaboration are:

- System architects have an overview of existing system architecture and can thus extrapolate their knowledge to new projects. This enables rough effort estimates and a statement whether the development roadmap looks realistic in terms of technical feasibility and time to completion.
- System architects can analyze high-level dependencies between projects, like: "Subsystem X only runs if subsystem Y is installed". We better complete the first assemblies of subsystem Y well before the first scheduled prototype test of subsystem X, or alternatively have to build an emulator of subsystem Y."
- System architects need to plan their own work, which is usually feeding into the different development projects on the development roadmap.

Let us elaborate on the last aspects of work planning for system architects. Each project means a certain workload for system architects. We saw in Section 12.6 that one can make a prediction of the workload for a given project (Figure 12.3). The roadmap enables the creation of an overview of workload over time, which may allow for optimizing resource allocation and planning in each of the development projects, with the aim of not exceeding the total capacity of system architects. This is particularly relevant in organizations in which system architects come from one team or department with a defined head count.

Figure 12.5 shows an example of such a workload overview across projects. Some of the shown workload curves are similar to the one from Figure 12.3 in Section 12.6. They are, however, not completely identical to that curve. The reader can easily verify that the total workload for system architects would have high peaks at certain moments, in case the timing of projects or the shape of the different workload curves in Figure 12.5 changed.

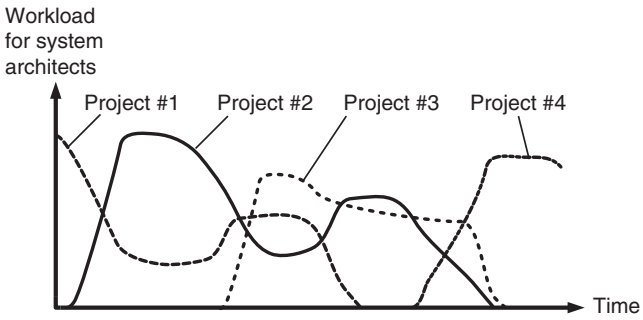


Figure 12.5 Using workload predictions from Figure 12.3 to support planning based on roadmaps.

If we assume that the average workload of project #1 to #4 in Figure 12.5 is perfectly matched to the capacity of available system architects, then the figure shows a close to ideal situation, in which peaks in total workload are moderate because peaks and valleys of the individual projects' workload curves level each other out. For example, the first peak load in project #2 coincides with a valley in project #1.

Reaching a close to ideal situation as described hardly happens by itself. Only by planning based on roadmap information can the system architects avoid capacity overflow or underflow. The easiest means of optimizing plans is to start tasks earlier than necessary or to postpone them if feasible with an acceptable risk. However, there may be cases in which it is impossible to resolve bottlenecks with planning only. The most easy way out in the long term may be to add resources, but this neither helps immediately, nor can it always be done in practice. Then two possible actions to consider are:

- Suppress certain tasks or reduce the ambition level of them. This results in a risk that should be addressed via risk management as it has been discussed in Section 12.6.
- Escalate the bottleneck to the roadmap planners. It may be that other disciplines have bottlenecks as well and that roadmap planners have to admit that their roadmap is unrealistic at the end. This will then ideally result in a modified roadmap.

Table 12.6 describes the cooperation between system architects and people in roadmap planning as a win-win situation.

12.9 Production and Distribution

Yassine and Wissmann [277] point out the strong interrelation between product architecture, assembly, and distribution: The product architecture enforces certain assembly steps and enables or inhibits the delay of assembly to entities inside the distribution chain. This is why production and distribution people are important stakeholders in system architecture.

There are cases in which the production and distribution environment is not defined completely during product development, but is already present to a certain extent when development starts. In such a case, it may be a constraint on the system architecture that certain production or distribution steps have to be carried out using given elements of an existing production and distribution environment. Such constraints are ideally captured by the requirements engineer and can be refined in the dialogue between the system architect and the production and distribution people.

Table 12.6 Close collaboration between system architects and roadmap planners as a win-win situation

What roadmap planners give	• Transparency about the roadmap creation and about roadmap changes • Willingness to take system architecture knowledge into account
What they get in return	• More realistic roadmaps
Obligations toward system architects	• Roadmap planners provide early drafts of roadmaps and inform about later roadmap updates
What they can expect from system architects	• System architects make rough feasibility studies based on roadmap input • System architects optimize plans in system architecture based on roadmaps • System architects escalate capacity bottlenecks that cannot be overcome by optimizing system architecture plans alone

Table 12.7 describes the cooperation between system architects and people in production/distribution as a win-win situation.

12.10 Suppliers

Suppliers provide deliverables that usually have interfaces with the system-of-interest or even inside it. The work with suppliers thus always involve interface specifications. Since system architects are the owners of interface specifications, they should be involved in the interface agreements with suppliers, who best also make their respective architects part of the dialogue.

Table 12.8 describes the cooperation between system architects and supplier's architects as a win-win situation.

12.11 Marketing and Brand Management

Yassine and Wissmann [277] have made a thorough analysis of the relationships between product architecture and marketing as well as brand management, among others. They come to the conclusion that product architecture influences the firm, thus the organizational entities that we would call stakeholders. Additionally, they point out multiple open research questions exist toward the "consumer perspective regions." We can thus on the one hand consider marketing

Table 12.7 Close collaboration between system architects and production and distribution people described as a win-win situation

What production and distribution people give	• Insights into possible assembly scenarios • Insights into the assembly possibilities inside the distribution chain
What they get in return	• Products that are designed for manufacturing • Products that are designed for optimized distribution logistics
Obligations toward system architects	• Production and distribution people explain the processes they are responsible for • Production and distribution people evaluate cost of different manufacturing and distribution scenarios
What they can expect from system architects	• System architects account for manufacturability from the first moment of systems architecting • System architects take special needs of distribution or decentralized finishing of the manufacturing process into account

Table 12.8 Close collaboration between system architects and the supplier's architects as a win-win situation

What the supplier's architects give	Technical insides from the supplier side that are needed for making well-founded interface agreements
What they get in return	Technical insides from the customer side that are needed for making well-founded interface agreements
Obligations toward system architects	The supplier's architects strive for a technically sound interface agreement
What they can expect from system architects	The system architects strive for a technically sound interface agreement

and brand management as potentially important architecture stakeholders but on the other hand not simply answer the mentioned open research questions. Therefore, we will limit ourselves to providing hypotheses based on our own experience.

Quality requirements are an important input to systems architecting, as it will be discussed in Section 14.1. We believe that the marketing and branding strategy of the organization provides important quality requirements, because it can answer how important criteria like easy product differentiation, short time to market and the possibility of radical innovation are considered to be for sustainable success

and growth on the market. The derived quality requirements may be important for the long-term success of a chosen architecture, whereas requirement-based making of architecture decisions only focuses on currently identified requirements. Since marketing people are usually focused on current market needs, the system architect should explain well that the life time of the system architecture is. It may need to take future market development into account. The architect should also be aware that nobody can predict the future. By making statements about the future, marketing will do guesswork and may change their mind whenever future market needs require this. The architect should always be prepared for change. The dialogue with marketing people helps them to develop a sense for the needed flexibility over time, and architects should use their own sense of the matter to make architecture decisions that reach further out into the future than marketing will be able to reach with their predictions.

The architect should also be aware of the very different fields. Marketing and branding people will not speak the same language as system architects. The system architect needs to explain the own needs well and should ask the right questions (for example: "Is it more important to achieve strong differentiation of products or fast time to market?"). The system architect should also be aware that particularly marketing uses a language that may seem strange to people with a strong focus on the solution domain. This has multiple reasons, one being that marketing has to influence potential buyer's emotions, whereas the human factors usually considered in systems architecting are the ones important for human machine interaction.[2] As goes with each dialogue, the one with marketing and branding people will only succeed if there is recognition for each other's work. In order to initiate a successful dialogue, the system architect needs to accept that a more emotional and less technical approach to the system-of-interest is necessary for business success and is chosen by the marketing people on purpose and not due to lack of technical insides.

Table 12.9 elaborates on the above hypothesis by describing a potential win-win cooperation situation.

12.12 Management

Fried and Hansson write in their book "Rework" [82]: "if you're opening a hot dog stand, you could worry about the condiments, the cart, the name, the decoration. But the first thing you should worry about is the hot dog" (p. 72). Applied to

2 It would be a very nice discussion to assess whether the user's emotions need more focus in human factors engineering. This question is beyond the scope of this book, but we direct the interested reader to the article "Engineering Joy" [100], which we have already made reference to in Chapter 3.

Table 12.9 Close collaboration between system architects and marketing or branding people can be a win-win situation

What marketing or branding people give	Willingness to face a more technically-oriented conversation partner than usual
What they get in return	A system architecture that will support their strategy instead of just the next product on the roadmap
Obligations toward system architects	• Marketing and branding people explain the marketing and branding strategy • Marketing and branding people answer the system architects' questions that help the identification of quality criteria for the system architecture
What they can expect from system architects	• System architects accept that marketing is needed for business success, even if it speaks a language that sounds strange to some engineers • System architects take the marketing and branding strategy into account in the daily work on system architecture

systems architecting, this could mean: The first thing system architects should worry about is system architecture. Of course, a system architect who has never facilitated an interface agreement and never shared any architecture ideas with others is like a hot dog seller without hot dogs. Still system architects need to communicate their work results not only to the development organization for ensuring that the realized systems follow the intended architecture, but also to their management stakeholder in order to ensure that the value of systems architecting keeps being recognized on management level – and maybe also to obtain management directions, in the context of the architecture governance process.

Since most organizations do not sell architecture, the architects will need to justify how they contribute to the turnover of the organization. Chapter 3 explains this and also gives hints on which values to communicate. It is important to communicate the value of systems architecting and the reasons to believe in return of investment to management on a regular basis in order to maintain support for architecting activities in the organization.

Depending on how "high level" the management likes to be addressed, it can be sufficient to communicate and explain the value as described in Chapter 3 – best accompanied by concrete success stories from daily business that underline how system architects created value and return of investment for the organization. For the more technologically interested management stakeholders, it may, however,

be as well important to show architecture deliverables like a piece of architecture documentation, again best associated with a success story indicating, e.g. how this concrete piece of documentation helped systematically analyzing possibilities to reduce cost. Model-based systems architecting can assist the latter kind of communication, because it supports the creation of stakeholder specific views that can present model contents in a representation suitable for management.

It is our experience that effective systems architecting requires management awareness, even better: management commitment. On the way to get there, we have observed in organizations without well-established architecture practices that inofficial architecture activities by few engineers can produce first success stories of applying methods in systems architecting, which will then serve as show cases for motivating the official introduction of systems architecting via a change project. In such early activities, it is particularly important to focus on "quick wins," i.e. activities with low effort that can create high value that is visible to the organization.

Since management trust in the system architects is very important, system architecture people with the possibility or even the obligation to occasionally present status to management should reserve enough time for preparing and doing the communication to management.

Our three golden rules to consider if you are an architect who is about to meet management:

- When presenting problems always come with a solution proposal – or even better: an evaluation of several solution scenarios and a recommendation which one to choose.
- Use simple communication and do not expect management to have time for a developing a deep understanding of your area of expertise. Rather be aware that they can expect you to make them understand what they need to know in simple, short statements. If you are a passenger in a car and the driver has not seen a pedestrian, you will shout "watch out for the pedestrian!" and not make a talk about the medical impact of car accidents on pedestrians. Management is the driver, and it is your task as a system architect to make them aware of information that is relevant for their decision on how and where to drive.
- Never use a good relation to management for gaining more authority over architecture stakeholders. The system architect's influence in shaping the system-of-interest comes from the system architect's ability to convince others with well-founded reasoning and not via management dictate. Always solve a conflict with a stakeholder in personal dialogue with that stakeholder and never use management as the mediator of your conflicts. They can expect you to manage your own conflicts.

Table 12.10 A good relationship between system architects and management as a win-win situation

What management gives	• Trust in the return of investment that will be generated when investing into good systems architecting • Input that helps setting directions for system architecture (e.g. via the architecture governance process)
What they get in return	• Predictable projects • Better products (see Chapter 3) • Better communication between engineering domains
Obligations toward system architects	• Management is patient regarding return of investment
What they can expect from system architects	• System architects explain the value of systems architecting (see Chapter 3) • System architects provide "bird perspective" information about the system-of-interest instead of overwhelming management with technical details

Organizations that have implemented an architecture governance process (see Section 14.1) may also establish a collaboration between system architects involved in that process and managers providing the corresponding management input. Table 12.10 describes the good collaboration between system architects and management as a win-win situation.

13

Roles

13.1 Roles

On systems engineering conferences, we have met people who provide their employers with requirements specifications, architecture descriptions, verification strategies, and many more value-adding deliverables. When we looked at their business cards, then some were "Digital Signal Processing Engineer," some may have been something like "Head of Mechanical Design Unit F," and others were "System Architect."

No matter what is written on these peoples' business card, if we meet them on a Systems Engineering conference then it is probably because they are applying thoughts, methods, or processes that are attached to Systems Engineering. If we meet them in a session about system architecture, then they are probably even mentally involved in systems architecting.

In the development of a system, we may ask who currently performs a systems architecting activity. The answer should be independent of this person's position in the organization, but it should be based on the kind of activity that different individuals are currently carrying out. When talking about the activities in systems architecting and the skills and competences needed to carry them out, then it is possible to describe them independent from the shape of the organization, which makes the description of systems architecting reusable across different organizations and robust against organizational changes.

To avoid the need to consider different kinds of organization, the notion of a *role* is being used here. A role is an idealized mental representation of a worker or team that has to carry out a certain set of activities. A role usually goes along with a *role description*, defining the objective of the role and tasks, responsibilities, and competences that are related to the work to be done. Like an actor can play a the role of a president without ever having been elected, a role can be assigned to a person who has not or not yet been given the corresponding position in the organization. This chapter will focus on roles without looking at the organization.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Considerations about systems architecting in the context of the organization can be found in Chapter 22.

Persons with the system architect role are central in system architecture. They have to ensure that system elements fit together during system integration and together act as a whole that satisfies the system requirements. They take care that proper architecture and its description are in place and understood in the organization. *System architects* in the terminology of this book are the people having the system architect role. Some companies may give a different name to this role, and often, the people filling the role will have other roles in their daily work. This chapter describes what people do and how they behave when they act as a system architect or as a member of a system architecture team, and what skills they need to do so.

13.2 The System Architect Role

13.2.1 Objective

The objective of the system architect role is described in very general terms here. It will be strongly based on the system architect's interaction with stakeholders, as described in Chapter 12.

- The system architect ensures that the system architecture of the system-of-interest will follow applicable directions (for example, the architecture directives from the architecture governance process).
- The system architect ensures that the system architecture of the system-of-interest is consistent and satisfies the system requirements as well as potential quality requirements.
- The system architect ensures that the system architecture is the result of a correct decision-making process.
- The system architect ensures that the system architecture description correctly describes the system architecture as followed by the system of interest and that its stakeholder-specific views are understood by the corresponding stakeholders.
- The system architect ensures that the system architecture is agreed with the appropriate stakeholders via the system architecture description (key objective: agreed specification of interfaces).

13.2.2 Responsibilities

The system architect is responsible for

- Identifying stakeholders, their concerns, and viewpoints to define the views needed to address the concerns.

- Creating the system architecture description with appropriate perspectives and views.
- Identifying stakeholders who need to review the system architecture description.
- Establishing traceability between the system architecture description and the system requirements or use cases and the discipline-specific architectures like software or mechanical architecture.
- Conducting reviews with the identified stakeholders and ensuring that the system architecture satisfies the requirements or enables the use cases it is traced to.
- Providing stakeholders with expertise about the system architecture and the work on it.

13.2.3 Tasks

The system architect has a multitude of tasks. In order to find out whether a task to be done is one to typically assign to the system architect, one can check if at least two of the following criteria are met:

- The task is multidisciplinary
- The task addresses a system-level solution
- The task requires defining or analyzing interfaces as seen from one or more perspectives of the system architecture description, for example:
  - Cross-subsystem interfaces of the physical perspective
  - Interrelationships between functions of the functional perspective
  - Cross-layer interfaces in the layered perspective

Here is a non-exhaustive list of typical tasks for the system architect:

- Contribute to feasibility studies and effort estimations
- Help project leaders breaking down the work in projects
- Contribute to project planning and verification planning
- Find solutions together with technical experts
- Ensure each interface is agreed with stakeholders on both sides of the interface
- Create the system architecture description
- Publish and communicate the system architecture description
- Explain the system architecture, based on the system architecture description
- Proactively identify cases in which more explanation of the system architecture is needed in order to enable stakeholders to make correct decisions in their area – and ensure that the corresponding explanations happen
- Answer stakeholders' questions, based on the system architecture description
- Improve the system architecture
- Assist in interpreting system-level test results (e.g. to state a hypothesis in which subsystem to look for the cause of a failure during system verification)

Readers interested in a very vivid description of what system architects do should read the work by Gerrit Muller [178, 179].

13.2.4 Competences

Here is again a non-exhaustive list of competences for a typical system architect:

- The system architect has the competence to make architecture decisions as long as they do not affect schedules or budget of other stakeholders.
- The system architect has the competence to lead negotiations about architectures decisions in which the stakeholders need to be involved, for example because their schedules or budget are affected by the decision.
- The system architect has the competence to approve the system architecture description.

13.2.5 Required Skills of a System Architect

The main skills of the system architect are systems thinking and abstraction as well as leadership and communication skills. Communication skills are needed because the system architect will need to interact with the stakeholders of the system in order to make the right system architecture and in order to ensure that it is understood and discussed in an understandable language by those that need to understand it. Leadership skills are needed, because the right architecture decisions are not falling into place just by themselves and the system does not follow the right architecture just by itself. Someone needs to take the lead in ensuring that this happens and that conflicting stakes around the system are handled with respect and yet with a direction of moving forward toward one consistent architecture. Consistency needs to be obtained across all stakeholders authorized to be interested in or to contribute to the realization of the system. The leadership and communication skills are so important that they are also discussed in Chapters 15 and 23. In the current chapter, we will therefore not elaborate further on these key skills of the system architect and will rather go on to explaining the abstraction skills.

Abstraction according to the original Latin roots of the word is about drawing something away. In this case, we draw the essence of a concept out of all the non-idealities of the concept's materialization in a real world. This is shown in Figure 13.1: The system architect in the picture is looking at a relatively complex system, which had to be built in a very complicated way due to whatever constraints. Due to her abstraction skills, the system architect does not think about the system in the complicated way in which it appears. She rather grasps the essential principle of operation, which is "C flows from A to B." The figure has been chosen

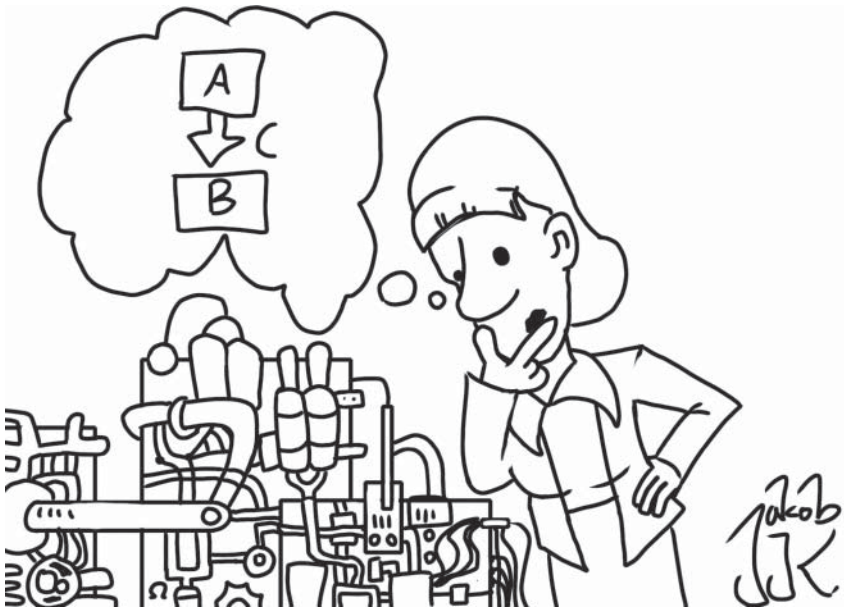


Figure 13.1 Abstraction skills. Source: © 2015 Jakob K., reproduced with permission.

to be very close to the mentioned literal meaning of "dragging something away." The system architect in the picture drags the principle of operation into her mind by looking at the concrete representation from the system.

Even though abstraction is thus based on the concrete reality, no one prevents us from making an abstraction of a concrete system that has never been built so far. This is the main strength of abstraction in systems architecting: By finding an abstract representation of a system to be built, the system is reduced to its essentials. These can then be taken care of with special focus during the development of the system, even before the first implementation of the system exists.

A system architect with abstraction skills can

- accept descriptions of system behavior that exclude the "pathological cases" and can thus generate simplified descriptions of the system
- record very complex interrelationships and still see structure in the system

There is also a related skill, which we would like to call "reverse-abstraction": Once the system architect has derived conclusions from an abstract model of the system, these have to be translated back into the concrete world. In the case of a marketing stakeholder, this could mean: The system architect has to explain the consequences of an architectural constraint on the way the product is perceived by its customer. In the case of an engineering stakeholder by contrast, this could

mean: The system architect has to explain the consequences of having chosen pattern *XYZ* on the way the information has to be encoded before being transmitted via a wireless link.

The system architect of course needs more skills than mentioned so far. Some of them will be discussed in Section 13.6. More of them can be found among the skill sets typically expected from every systems engineer. A comprehensive discussion of these skill sets can be found in the INCOSE Systems Engineering Competency Framework [124].

13.2.6 Required Skills for Model-Based Systems Architecting

One may think that knowledge of modeling language and modeling tools are the most important skills in model-based systems architecting. Of course these are helpful, but the more important precondition for being successful with model-based systems architecting are, in our opinion:

- The already mentioned abstraction skills, because all models are abstractions of reality
- Awareness about the "single source of truth" paradigm (see Section 11.11.1) and the ability to create followers for it in the organization
- Understanding of the separation between view and model (see Section 9.7)

13.3 System Architecture Teams

There is often more than one person with the system architect role. Several different system architects can work on the same architecture description, for example by working on a common architecture model. They should ensure coherence of the system architecture by working in close collaboration. They should work together as a system architecture team, which is a role of its own. The team should ensure that team building happens and team rules and processes are agreed. It should also have means of prioritizing and processing common topics of the team, like, e.g. a joint approach to architecture enablement in the organization. The system architecture team links system architects that work in different areas or activities. For example, each of them can be allocated as the system architect in a different feature team (e.g. Larman and Vodde [155]).

Via the system architecture team, the system architects can aim at ensuring the following:

- They follow the same methods and best practices.
- The system architecture stays coherent.

- The way of working with stakeholders is consistent such that systems architecting becomes a predictable and consistent activity for stakeholders who collaborate with more than one system architect.
- There can be a second opinion on difficult questions.
- More than one system architect understands the system architecture such that a member can be substituted while absent.
- The workload generated by certain activities can be spread to more than one person.
- The system architects can help each other.

There are multiple ways in which system architects can help each other, apart from giving each other assistance on all different kinds of details during daily work:

- People with understanding of different engineering disciplines can join their understanding during the team work, such that multidisciplinary understanding is reached in the team.
- People with different strengths and weaknesses can use their strengths to compensate for others' weaknesses.
- People with different approaches to the same task can find a better approach by choosing the best elements from each of the approaches.

When staffing a system architecture team, one should keep in mind to mix people whose strengths and weaknesses can compensate for each other and whose approaches are different in order to increase the likelihood of finding a good approach for each task. For example, one should mix pragmatic people with highly analytical people. Reasons for this are:

- A team staffed only with analytical people can be trapped in too thorough analysis that does not stop when the solution suffices.
- A team staffed only with pragmatic people can be trapped in producing inconsistencies or chaotic results.

In very small organizations, the total head count limits the number of people that can participate in a system architecture team. But even in larger organizations, there is a limit to the number of persons who can efficiently collaborate as a system architecture team. When we saw winning system architecture teams in real organizations, they were often staffed with exactly six people. We do not have well-founded explanation why a team of six people would be the optimum winning team; however, we can think of the following trade-off regarding team size:

- To be as multidisciplinary as possible, the team should consist of people with a background in ideally all the different fields of engineering that are needed for the development of the system-of-interest.

- In order to still be able to focus on the rest of the team, each team member is potentially limited by the own "seven ± two rule" [171], which means in the "minus two" case that there is a limit of five people to focus on for each team member. Counting also the team member from whose perspective this is counted we come to a total of six team members.

Once a system architecture team has been created, it can start advancing the practice of systems architecting in addition to sharing and doing the operational work.

13.4 System Architecture Stakeholders

We have seen typical system architecture stakeholders in Chapter 12. Some of them have dedicated roles in systems architecting. For example, requirements engineers are the ones delivering the requirements input. Others will become relevant in case a procedure states to "involve the stakeholders." For example, there may be a procedure stating that a certain piece of architecture description has to be reviewed by the stakeholders. Which stakeholders to involve is then dependent on the content of this architecture description. If it has focus on production, for example, then the system architect may need to involve production people, whereas verification people may need to be involved in cases in which the test access points of the system are under review. In the descriptions of certain systems architecting activities like the review of a piece of architecture description, it is therefore sensible to use the generic role "stakeholder" instead of a specific role like "production infrastructure engineer."

13.5 Recruiting System Architecture People

As stated earlier, this chapter is about roles and not about positions or job titles in an organization. When we talk about "recruiting," then we talk about spotting people who are the right ones for the system architect role. Whether they have to be hired first or just found inside the organization does not necessarily matter.

It is an observation that key players in development are often recruited as system architect. This is of course due to the fact that these persons have the overview of at least parts of the system, and creating overview is one of the central tasks of the architect. The question is whether the technical skills that most system architects have are their most important skills. After having stressed that communication and abstraction skills are the most important ones for the system architect, the answer we give is a "no, but …."

A system architect needs product understanding and technical skills to interface to technical experts, who will be most comfortable in expressing themselves in technical terms only understood by another expert. A system architect also needs to spread the ideas of new architectural concepts and therefore needs credibility and in consequence trust among other engineers. Technical excellence is one key in being credible among other technically excellent people. Yet, a system architect who is heavily involved in making system architecture cannot at the same time keep up with the state-of-the art of the technologies of the system-of-interest. The system architect will rely on other experts to provide up-to-date technological information, as far as it is relevant to system architecture.

Even more important than the skills are the right talents and the right mindset. Systems architecting needs people with a multidisciplinary mindset and the ability to carry thoughts into different disciplines. These are also known as "t-shaped persons" or "t-shaped individuals" (Stickdorn and Schneider [235] p. 111; Kelley [138]. p. 75), because they have a deep understanding of one area and the willingness to think multidisciplinary, which leads to a graphical visualization in the shape of the letter "T" (the vertical line of the letter represents the deep understanding in one area and the horizontal line represents the multidisciplinary thinking).

Due to the need for systems architects with a good network to the stakeholders, the needed technical and product understanding and credibility among engineers in different areas, a good place for recruiting system architects is your own organization. Try to find the T-shaped people. Look for those people who are always called by others in case a multidisciplinary problem needs to be solved. Look for those whose absence makes people nervous during such problem solving.

Also look for people who often expose themselves by trying to make things better. Not the ones who can complain about everything and postulate that it can probably be made better. Choose the ones that actually initiate improvements or at least always have a proposal for improvement ready. Leandro Herrero calls these people "simply healthy restless people […] with a mixture of […] frustration and at the same time commitment to make things better" ([101], p. 282). He does so in the context of looking for change agents. Actually, one can say that systems architecting requires people with change agent skills, for example because ensuring that stakeholders stick to an interface agreement may mean that they have to *change* the solution idea toward one that complies with the interface definition, but also because establishing a multidisciplinary mindset sometimes requires the initiation of a mindset *change*.

Finding the right people is more important than looking for people with the right position in the company. For example, people with a good sense for project management are not necessarily good architects, because they have a focus on timing rather than on the right concept and a common understanding (see

Figure 12.2) – and actually one should try to avoid assigning the project leader role and system architect role to the same person, because the roles have to act in different directions (Figure 12.4). Still, also a system architect needs the ability to finalize tasks according to schedule instead of getting lost in details.

System architects are ambassadors for the system architecture in the different areas of engineering. It is thus good to look for "inofficial leadership" in these areas: A development engineer who is often invited for technical discussions based on wishes from the other participants is a good candidate for becoming a system architect. The fact that other team members wish to have this person in the room when discussing certain problems can be a sign that this person has facilitation and mediation skills and the ability to bring a solution forward – thus typical skills of a system architect. Of course, there may also be persons who are omnipresent because they have succeeded make themselves "indispensible" in the organization for instance by not sharing their knowledge. These persons are also invited to many activities for obvious reasons. They are bad candidates for becoming system architects, because knowledge sharing is essential in architecting a system.

Recruiting system architects within the own organization of course also brings some challenges:

- The people with the needed skills are often regarded as "key players," and their superiors may claim that it is impossible to hand over their work to others.
- Also when having the system architect role, these people may still get involved in their old work. Care has to be taken that this will not always win the battle for priority.

In situations in which such challenges are met, the following thoughts may be helpful: When a key player is removed from a team of development engineers and replaced by someone else, the organization becomes less dependent on "heroes" (i.e. the key players without whom nothing works). Furthermore, these persons will feel a career jump by being nominated as system architect and might take this as a reason to stay some more years with the company. Thus, the know-how of the former "heroes" stays in-house and can be handed over to others gradually.

When recruiting system architects, also think of the system architecture team they will be part of: which talents or skills are missing? Are there more team members with analytical approaches or more pragmatic people? Try to recruit architects that bring the missing talents, skills, and approaches into the team.

13.6 Talent Development for System Architects

Each of us can go to a dancing course, but some will have to learn dancing by practicing hard, others will just seem to dance by themselves. The latter ones are

of great talent for dancing. Assuming we have recruited people of great talent for systems architecting as described before, will they in future architect the system optimally based on their talent? Probably not. Just like a talented dancer can still learn from the right teacher, system architects also need to keep learning, partly from being trained in methods and tools, partly from gaining experience on the job.

When we have recruited a new system architect, we have most likely found a "t-shaped person": Apart from an existing background in an own engineering discipline, there is a talent for multidisciplinary thinking and as a consequence a broad knowledge. The system architect will now be trained in system architecture methods, processes, and tools, but also in other systems engineering practices, e.g. in the disciplines of the important systems engineering stakeholders like requirements engineering and verification. Furthermore, the system architect will learn on the job while doing coordination work between different engineering disciplines. The learnings will not only cover systems architecting practices, but also the specialities of different technologies and disciplines that are related to the subsystems. The system architect will gain different depths of understanding related to different subsystems.

Persons who have proven enough professional excellence in the system architect role may get tasks that aim at improving the systems architecting discipline itself rather than the system-of-interest. These people will thus have less contact with the operational engineering business than some of their colleagues. As a consequence, they will no longer be close to the carriers of up-to-date knowledge in different engineering disciplines, and thus, their own knowledge of these disciplines will become outdated.

If we summarize this development from a t-shaped person to an experienced system architect who is able to improve the systems architecting approaches of the own organization, then we see a development like it is shown in Figure 13.2: The figure shows the described development of the architect in three snapshots from the left hand side to the right hand side in diagonal upward direction. The vertical dimension in each of these snapshots symbolizes the depth of technical understanding in different technologies or subsystem-related disciplines. The horizontal dimension that is pointing away from the reader symbolizes the depth of understanding for different systems engineering disciplines, and the remaining horizontal dimension allows for showing different disciplines next to each other.

The left snapshot in Figure 13.2 shows the situation in which a system architect is often recruited: We see the "T" shape of a t-shaped person, which in general of course implies more than can be shown in the figure. Comprehensive knowledge in systems engineering will usually be missing in this stage, unless the person has been to a systems engineering education or has prior experience in systems engineering. In these cases, Figure 13.2 is not applicable.

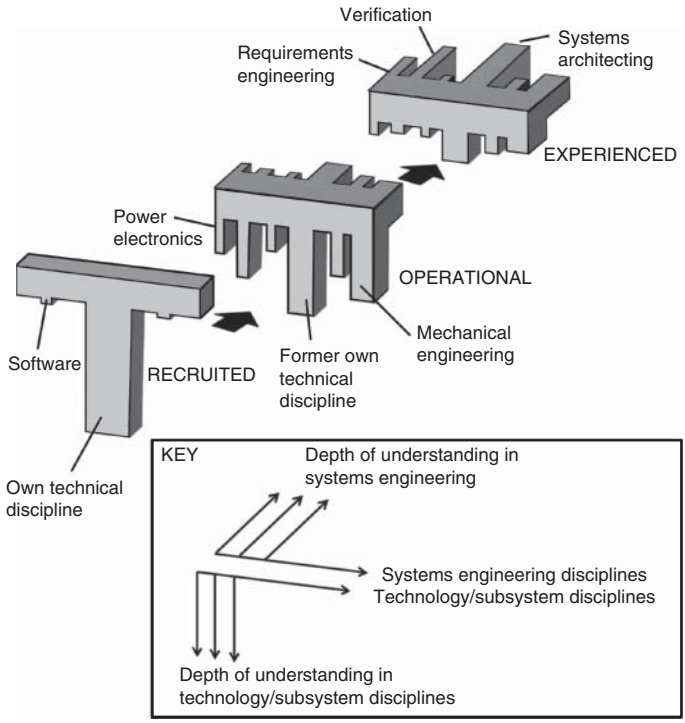


Figure 13.2 Changing profile of a system architect with growing experience, based on the assumption that there has not been any systems engineering career or education upfront.

The middle snapshot in Figure 13.2 resembles very much a profile shown by Maier and Rechtin [164], as it will be discussed soon. It shows the situation of a system architect who has become operational in the systems architecting work: Knowledge in different technologies, engineering, and systems engineering disciplines is present. The knowledge in the former own technical discipline has degraded. This does not necessarily mean that the system architect has forgotten the learnings from an earlier job, and it rather relates to the fact that technologies and the corresponding engineering approaches evolve and change fast such that everyone who is not following the related technical discipline with full dedication will no longer be up-to-date. For the system architect, this is the case for most engineering disciplines, due to a conflict of the required dedication with the new focus on systems architecting. The loss of proficiency in the own area of technology is an important aspect to consider for engineers who like to become system architect. Only when accepting this loss, they can gain

proficiency in systems architecting with full dedication. As already mentioned, Maier and Rechtin show a quite similar snapshot in their book "The Art of Systems Architecting" ([164], pp. 8–9). They trace it back to a lecture by Bob Spinrad at the University of Southern California in 1987. While they show the "*required* depth of understanding" in the different subsystem disciplines and omit the systems engineering dimension, we show the actually obtained depth, comprising systems engineering. The reason for us not to show the required depth like Maier and Rechtin was the preceding explanation about the loss of proficiency in the original engineering discipline. It would not be visible if we only sketched the proficiency actually required for being a system architect. Maier and Rechtin can use their representation to explain an other important aspect regarding the depth of understanding: It is not always required to understand each subsystem in detail; however, the system architect has to "dive in" in some cases and find out the very details of a task at hand. For this reason systems architects will gain deeper understanding in different disciplines.

The rightmost snapshot in Figure 13.2 shows the profile of the experienced system architect. Detailed subsystem understanding has now been replaced by understanding for the different systems engineering disciplines, first and foremost of course systems architecting.

Note that Figure 13.2 shows a view on a mental model, which is like every model an imperfect representation of reality. It is due to that imperfection of our model and due to the fact that the different facets of human talents and human personality are unlimited that we expect we could find excellent system architects all over the world whose career or current profile has nothing in common with the profiles shown in Figure 13.2. We have constructed the model to warn about the potential loss of technical proficiency in the system architect's former own area of expertise and to motivate the need for trainings in systems engineering methods and tools. However, we highly discourage the use of this model for talent spotting or for measuring the skill level of a system architect.

So which are the trainings we can use to accompany the even more important learning by doing of the system architect? Here is a non-exhaustive list for your inspiration:

- Leadership trainings
- Trainings in processes (not only in system architecture but also in requirements engineering, verification and validation, change and configuration management and in other systems engineering disciplines)
- Trainings in systems architecting methods (e.g. in the FAS method we will describe in Chapter 17)
- Trainings in a modeling language (e.g. in SysML)

- Trainings in the tools used for supporting the above-mentioned methods and processes as well as the modeling language, so for example a SysML modeling tool
- Communication and presentation trainings

Since we see on the job experience as the most important aspect of talent development for systems architects, we recommend to make a systematic planning not only of the training in methods and processes, but about their application in real-world projects. In the best case, both happens simultaneously, such that training contents become tangible in the daily work immediately.

14

Processes

14.1 Systems Architecting Processes

14.1.1 Overview

The systems architecting processes are the core processes for the system architect. They comprise processes for architecture definition according to ISO/IEC/IEEE 15288:2015 [115], which overlap with the processes called "architecture conceptualization," "architecture elaboration," and "architecture evaluation" by the standard ISO/IEC/IEEE42020:2019 [110]. The mentioned standard ISO/IEC/IEEE42020 also describes more strategic or management-level processes like architecture governance and architecture management. It furthermore postulates an architecture enablement process that ensures the necessary preconditions for systems architecting (skills of the personnel, infrastructure, organizational and project processes and management systems, etc.).

Table 14.1 shows an example how some very common inputs and outputs of the systems architecting processes and also how the contributing roles could be defined. One major output is the system architecture description (see Chapter 8). But also the learnings from carrying out the systems architecting work should be captured during the work and should be phrased as heuristics (see Section 9.11.1), e.g. in a best practice document. This leads to a living document to be updated and used across projects.

The actual process for producing the system architecture description consists of doing the architecting and validating or reviewing[1] and approving its output.

A very simple explanation of systems architecting is to translate requirements into the architecture of the system, including the allocation of functionality and

1 See Section 14.1.4 for a discussion of the terminology.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Table 14.1 Examples of inputs, outputs, and contributors for the systems architecting processes.

Input to the systems architecting process	• System context • Requirements • Use cases • Quality requirements • Heuristics, patterns, and good practices • Roadmaps • Domain knowledge
Outputs from the systems architecting process	• Architecture description • Updated heuristics and best practices
Contributing roles	• System architect (drives the process) • Stakeholders (provide input and consume outputs)

performance to subsystems. This would assume the existence of a complete requirements specification. However, in many cases, one may need to collect the most architecturally relevant requirements before the other ones, because:

- The requirements may not yet be completely elicitated at the time at which the very basic architecture of a system needs to be decided on.
- Requirements may change during a project, e.g. based on new market inputs.
- The system architecture may have to be reused across multiple projects, whereas a requirements document often has one single project as its scope.
- Maintainability and scalability may be needed in later phases of the life cycle.

This is a reason for inputs like the quality requirements, as it will be explained further below.

The system architect has to ensure that the system architecture considers each life cycle stage of the system and the system's future evolution. In our experience, most system architectures live longer than they were intended to live. Therefore, the architect has to consider how to make the architecture endurable, which may be requested by one of the quality requirements we see as an input in Table 14.1. Here is a discussion of all the inputs in the table:

- The system context provides the scope for the system requirements, the use cases, and the system architecture. See Section 10.2.2.
- Requirements and use cases specify the expected functions, behavior, and performance of the system. The system architecture has to realize the satisfaction of the corresponding expectations.
- Quality requirements give guidance on what to optimize for (e.g. flexibility vs. simplicity, modifiability). These can be obtained in close collaboration between systems engineers and their stakeholders. Ideally, the requirements engineering

process provides quality requirements. However, we recommend that the system architect is involved in their definition. The high importance of the quality requirements results from their meaning regarding the life cycle of the system architecture: While most requirements aim at a certain product portfolio that is known at the time of the requirements elicitation, quality requirements drive architecture decisions that are difficult to change during the whole life cycle of the system architecture. Since fundamental architecture decisions are often valid for more than one release, it is important to look ahead of the releases that are usually in focus of the requirements engineering processes. The system architect should ensure that the decision on quality requirements is based on sufficient understanding of their impact on current and future systems. This typically requires direct interaction with the stakeholders. It is therefore very important that close communication between the system architect and the stakeholders happens. The system architect can prepare the questions that need to be answered by defining the quality requirements. Once quality requirements are defined, they can be used in architecture assessment (see Chapter 21).

- Heuristics, patterns, and good practices can help solve particular challenges of the given business. Typically, they are maintained throughout many projects and updated with the learnings from each project.
- Roadmaps can give an idea about the scalability requirements.
- Domain knowledge complements requirements in creating understanding about what has to be developed.

There are outputs that are not shown in the table. These are, for example, strategic information like the so-called architecture directives, which can be defined during the architecture governance process. The architecture governance process will ensure the correct direction and long-term orientation of the systems architecting activities with a scope that is larger than a single system development activity, for example: governance of architecture across multiple releases, across multiple systems from the same development organization or across a whole enterprise. Its implementation and application depend on the specific organization you target. This book therefore cannot provide general guidance. For inspiration, we recommend the standard ISO/IEC/IEEE42020:2019 [110].

After having made a very general description of the systems architecting processes, we will now show some examples of the process steps in architecture definition, first in a very general fashion and then with some relations to the contents of other chapters in this book.

14.1.2 Example of Generic Process Steps

Here are typical steps of the architecture definition process, in a very generic phrasing:

- Identify stakeholders and their concerns.
- Identify perspectives (these may be given by the standardized architecture framework used in the organization).
- Identify views for the stakeholders.
- Identify quality requirements. A quality requirement could be: "A new color variant of the system must be ready for sales three months after the start of development."
- Evaluate architecture alternatives based on the quality requirements and decide which one to choose. For example, to stick with the quality requirement of fast color change, a modular architecture may be chosen with a strict separation of colored housing parts and those parts that contribute to core functionality of the system (according to the pattern "separate stable from unstable parts" in Section 9.5).
- Model the architecture by information of different views and mapping the different views.
- Produce the architecture description by generating the views.
- Validate or review and approve the architecture description.

14.1.3 Example of Concrete Process Steps

Here is a proposal on how to use concepts from this book in the context of the systems architecting process (it is highly recommended that you taylor it to your personal needs by removing some steps or adding some):

- Use stakeholders from Chapter 12 as a default initialization of the stakeholder list, and assess whether the given project needs additional stakeholders.
- From Chapter 11, select the system context, the functional perspective and physical perspectives as well as traceability perspectives.
- Ensure that the system context description is available.
- Ensure that the base architecture meets the quality requirements, based on architecture evaluation (e.g. via an architecture assessment as described in Chapter 21).
- Use the FAS method (Chapter 17) to fill the functional perspective, based on use cases from the input of the process.
- Make functional views for the different stakeholders with their functions of interest and the functions having functional interfaces with them (see Chapter 11).
- Do the functional-to-physical mapping together with the stakeholders, based on the requirements.
  - Formal alternative
    * Find alternative mappings.

    * Use trade studies/decision theory (Section 20.4) to choose the optimum alternative. Use the requirements as selection criteria and ensure that particularly the non-functional requirements get enough focus.
  - Informal alternative
    * Assemble a team with representatives of all physical blocks that are potentially involved in the mapping [151]. In some approaches, this is called a feature team.
    * Give the requirements to this team and ask them to come up with a good functional-to-physical mapping and a reasoning why they chose the mapping.
- Generate the architecture description from the produced models, views, decisions, and reasoning (e.g. using the template from Section 8.3.2).
- Validate or review and approve the system architecture description according to Section 14.1.4.

14.1.4 Validation, Review, and Approval in a Model-Based Environment

Validation is a term with multiple definitions in different standards for different industries. Sometimes, it refers to an activity that confirms if an architecture description document satisfies the requirements, e.g. by means of a review. Sometimes it refers to an activity that confirms if a sample of the product meets the stakeholder requirements when being used in its operational environments by sample user. Sometimes it can be used for both or even for different kinds of activities. Since we like to address readers in different industries, we are not using the term any further in this section.

This section is about the confirmation that the contents of the architecture description are correct. There are several major aspects to this:

- The system architecture as described by the architecture description is sufficiently aligned with the long-term strategy of the organization, which may have been expressed, e.g. via architecture directions or quality requirements.
- The system architecture as described by the architecture description satisfies the system requirements.
- The system architecture as described by the architecture description is technically sound.
- The architecture description is consistent in itself.

Again, there are multiple versions of processes for achieving this. Let us make a very simple example of a process, in order to use it for discussing the details of reviews and approval in model-based systems architecting:

1. The system architect identifies the stakeholders that need to be involved in the review.
2. The system architect conducts a review with the identified stakeholders.
3. Potential findings from the review are addressed and the process is restarted, if necessary.
4. As soon as it is appropriate to confirm that the contents of the architecture description are correct, the system architect declares the architecture description *approved*.
5. The above process is documented, e.g. by listing persons involved and their roles, dates of reviews, actions taken to address review comments, and measures taken to verify that they have been done. It may also be appropriate to state which reviewers were involved in the creation of the architecture description under review and which ones were independent reviewers.

Some of the above steps may be replaced by the architecture assessment steps described in Chapter 21.

The above process may be too simple to be applied in industries or around systems with regulatory requirements toward processes or products. We recommend to the reader to retrieve information about the applicable regulations before adopting any steps of the above process.

In a document-based environment, one can simply follow the above process if it complies to the applicable regulations. It can be conducted on the basis of an architecture description document, and then, the outcome can be documented inside the document under review. This is of course an example, and dependent on the applicable processes or even regulations the statement may look different and an electronic or hand-written signature might be required, for example by means of a validated tool.

In a model-based environment, there is in the first place no document that one can review, approve, or sign. This has consequences for the approval process but also for regulated processes that may require a signature. For this book, we cannot consider the regulatory aspect since it is strongly dependent on the context and on the applicable regulations. We will therefore continue with the simplistic assumption that there is no regulation to take into account. We remind readers working in a regulatory environment that it is always possible to export model views into a document, which is then processed according to regulations for document-based work.

The following theoretical considerations are about how to ensure that information which is retrieved from the model can be provided with a maturity state indicating whether the appropriate stakeholders have reviewed and approved it. They are based on a discussion that took place at the 2014 annual workshop of GfSE (the German chapter of INCOSE) between one of the authors (J.G. Lamm) and

Professor J. Abulawi [3]. The starting point was the already mentioned notion to export model views as pictures and to approve them by signing them electronically or on paper. The notion would then be that views and model elements which are invisible in the exported material should be considered non-existing. Model-based document generation (see Section 8.3) was seen as the means of supporting the export of selected views in the form of a document. This approach can still be considered a fallback solution in case other approaches seem impractical. In theory however, a different approach could be chosen for proper model-based proceeding, based on the mentioned discussion [3]. This approach will be briefly presented in the following.

Based on the separation of view and model (see Section 9.7), let us assume that the model is the actual information carrier, whereas views on it simply lead to visualizations of the information. This means that the model itself should carry the information whether it has been reviewed and approved. However, since the stakeholders will access information from the model via stakeholder-specific views, these will be used for presenting the information under review.

How to get from a review via views to an approval of the model? First, it has to be ensured that each view contains the model elements that are relevant for the corresponding stakeholders. Rules should be stated, defining the criteria for a model element to appear on a view. For example, if a functional architecture model is under review by a feature team working on one functional block, the given functional block together with all functional blocks that have connections with it via ports could be in scope. In this example, the view for the given feature team would thus show blocks that are also in scope of a similar view for other feature teams. The same effect can occur via other perspectives. In Section 11.3, we showed different views from the physical perspective, which have a high amount of overlap regarding the model elements they show (Figures 11.6 and 11.7).

As we saw in the example, a model element can be in multiple views, which have to be reviewed by different groups of stakeholders. This means that one single review activity cannot come to the conclusion that a model element is approved. It can only come to the conclusion that the element is approved in the context of the view that was used for the review. We introduce the maturity-state *pre-approved* for views. It indicates that the review conducted on the given view has concluded to approve what has been reviewed.

Based on the pre-approval, automation can be added to automatically identify a model element as approved if all[2] the views in which it appears have been pre-approved. The pseudo-code below illustrates this:

2 Recent discussions [88] have questioned if "all" views have to be submitted to the described process. Consider that certain views serve a stakeholder called "model maintainer." They show, for example, whether there are model elements that use modeling language elements to soon be deprecated. The model maintainer will need to refactor these on elements on the long term, but

```
boolean isApproved (model element m)
   for each v in all views
         if v contains m
           if v is not pre-approved
              return false
           end if
       end if
    end for each
    return true
```

If it is required to produce approved visualizations, one may need to verify if a certain view is approved in order to automatically generate a maturity state to associate to the view.

The information given on a view is approved, if all the visible model elements are approved as defined above. The following piece of pseudo-code illustrates this:

```
boolean isApproved (view v)
    for each m in all model elements
        if m in v
           if not isApproved ( m )
               return false
           end if
        end if
    end for each
    return true
```

If the whole system architecture model should be approved by means of the above approach, then one has to ensure that each model element is contained in at least one view.

It is now up to the reader to consider if the above solution is practical. At least it requires automation and it also has a drawback: Model elements and views that were already in the approved state lose this state as soon as a new view is created, because it can be assumed that the view will not be pre-approved on creation, thus all model elements of the view can no longer be approved, and in consequence the same goes for all views showing those model elements. In theory, this is a healthy condition, because it reinforces systematic reassessment of affected descriptions on any change. In practice, this may be a quite heavy approach and more efficient means of impact analysis and reassessment may exist.

---

for the short-term review, they have no relevance. This may indicate that the corresponding view would not need to be considered.

14.2 Design Definition Process

The system architect will also be involved in the *design definition process* according to ISO/IEC/IEEE 15288:2015 [115] or may even be assigned as the driver in working according to this process. This may even bring system architects close to the work on discipline-specific architectures. The system architect may be able to derive outputs like budgets (see Section 20.5) or the detailed cutting lines and interfaces resulting from the partitioning of functions to physical subsystems.

Since this book is dedicated to the architecture definition process and other systems architecting processes, it will not elaborate further on the design definition. The outcomes of the architecture definition and design definitions processes overlap when delivering the description of the system design, because the system design refines certain aspects of the system elements that result from architecture definition. To expose this overlap, a brief example has been discussed in Section 11.9.1.

14.3 Change and Configuration Management Processes

The change and configuration management process in the scope of systems architecting ensures that

- Versions of the system architecture description or even of separately versioned parts of it are put under configuration management.
- Changes on approved system architecture descriptions are carried out with the needed focus on impact analysis and reapproval of the changed artifacts.

The discussion of the collaboration with the configuration manager in Chapter 12 provides more details on this topic. The system architect should consider to take the configuration manager role in case there is no configuration manager.

14.4 Other Processes Involving the System Architect

There are several processes in other disciplines that involve the system architects. The processes should ensure that the necessary cooperations between system architects and architecture stakeholders takes place. One such process is the requirements engineering process in which the system architect can, for example, assess the feasibility of satisfying a requirement with the envisioned or already given system architecture. Another example of the system architect's work in

the requirements process is to provide architecture input to the next level of requirements according to the zigzag pattern (see Section 9.1).

Chapter 12 provides a discussion of typical stakeholders in system architecture. The processes around them are typical ones that could also involve the system architect. Since this book is focused on systems architecting, those processes will not be discussed in detail.

15

Tools for the Architect

As a result of choosing provocative wording, the chapter title may create the expectation that this chapter will be about modeling tools, product data management tools, etc. This will not be the case. The mentioned software tools are just comfort factors that replace pencil and paper with their digital counterparts – like a desktop environment comprising a word processor and access to archive databases for a journalist. Excellence in being a system architect depends as little on the ability of using systems engineering software tools as the excellence in being a journalist depends on skills in operating a word processor software. This is why this chapter is not about the software tools like a modeling software – it is about the tools that really matter for being a successful architect. These are the correct mental approach, leadership, time management, and other helpers to accomplish what matters for the realization of successful systems. Together with the skills discussed in Section 13.2.5 these tools are essential for the system architect's success.

We ask the reader who expected an evaluation of software tools in this section to forgive us. Too often have we seen companies take shortcuts that ended up in purchasing software products instead of first defining what really matters. Especially in commercial businesses, it seems to fall very naturally to first define a high-level problem (example: system development is no longer under control) to then define an easy solution (purchase a software tool) and to then compute the investment case (the software costs less than the hassle arising from lack of control). There is nothing wrong to this approach if it is amended with the question "which mindset and approach and thus which people with new roles do we need," to be asked right after the problem definition.

Here are some tools for the system architect to be established before taking a software tool into use:

- The right *mind set* is the most important tool of the system architect. All others will sooner or later be established if this one is in place. Success comes to the system architect who applies reasonable thinking and keeps current deliveries,

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

future system life cycle stages, and future system releases in mind. System architects may have to deal with conflicting requirements, "political" statements and technically hard to solve problems. This is only possible with reasonable and mission-oriented thinking that will help make the "right" architecture decisions in such a context.

- *Leadership* was already mentioned in Section 13.2.5. The right mind set will be more fruitful if all system development follows the voice of reason, and followership requires leadership. System architects need to take the technical lead in system realization, but also work with different stakeholder groups teams and therefore need to be good at creating positive team dynamics (see also: Section 23.3).
- *Time management* will enable system architects to handle the fact that the holistic view on a system also comes with a holistic usage of the calendar. There are many detailed tools for time management described in the literature (e.g. [16]), and it is very individual for each person which ones work best. However, some time management technique will be needed for the successful architect, especially to ensure that "important" long-term work on future architecture directions and architecture enablement will not be a constant victim of "urgent" tasks at hand. Effective *priority management* considers both urgent and important tasks [221] – and then again, a stable approach to managing the balance between work matter and other important life matter needs to be maintained. For example, consider that system architecture is rarely created from a blank sheet. Usual systems have a history and therefore some legacy architecture that may have been state-of-the-art when it was first laid out, but is rarely the ideal architecture based on latest knowledge. A good system architect has a vision of the ideal architecture for the given system to fulfill the given present and future requirements. Still, architecture cannot be changed over night and thus the system architect will have to live with a nonideal architecture. System architects may therefore need the ability to happily live on with imperfect architecture, while in control of working on prioritized steps in the direction of perfection.
- *Facilitation techniques* for teamwork in presence and online [83] enable system architects to work on trade-off analyses and other systems architecting tasks with their architecture stakeholders. Many books describe techniques for facilitating workshops. A bit less literature exists on facilitation in online collaboration. Nevertheless, the principles are always the same: To ensure that people know what they can expect (prepare an agenda, define the expected outcome, explain what will happen with the outcome), and to ensure that people can both hear and see what is being discussed and concluded, in other words: Ensure that visualization is always being made, and that conclusions are captured and challenged until there is a common understanding. Ensure that people with difficulty in seeing or hearing what is going on are involved in the appropriate way.

We recommend to read about facilitation and to adapt the many things that are being written to what is appropriate for the given mission and the given audience. We also recommend to use visualization tools the facilitator is comfortable with. For example, it is inefficient to use an online collaboration tool that is most "en vogue", if one of the persons responsible for visualization during an online collaboration is not fluent in using it.

All in all, many of the tools for the architect are given by common sense, but sometimes it takes courage to give a voice to common sense. This chapter likes to encourage you to do so.

16

Agile Approaches

Anyone who has followed the countless discussions surrounding the term Agile in recent years, or decades, will have noticed that there are very different opinions about what Agile could actually mean. The Internet is scattered with various theories and propositions of what Agile is all about.

Some people take the view that a team or project organization is Agile if it uses an iterative–incremental process for its development process (We will discuss later in this chapter that this is basically not true!). Other people say that being Agile means brushing aside planning and all the other "old-fashioned project management stuff" and then just processing tasks by acclamation, depending on which stakeholder is screaming the loudest or how the product owner is prioritizing them. Still other people say that Agile means having lots of sticky notes on a wall, as well as regular stand-up meetings. There are a lot of myths and misconceptions about Agile.

In this Chapter, not only the term Agile will be enlightened and explained. We, the authors, are convinced that agile values and principles, as well as the practices, tools, and some of the frameworks that exist in this context, have many advantages that can be a benefit for any systems engineering project. We've seen organizations that have adopted agile principles, practices, and tools partially in their development units or even on a more broader basis.

However, so far we have hardly seen any organization in systems engineering that described itself as "fully Agile" and also present this to the outside world. The reason for this is that it is essentially about people, and only secondarily about processes or tools. Establishing an agile corporate culture means that there must be a fundamental willingness to cut off old pigtails and break new ground. Agile is about a change that goes deep into the culture of the company. It's about a new leadership mindset. Classic organizational structures from the long-gone industrial age are being called into question. To boil it down: Transforming

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

an organization with a classic and hierarchical management structure into an adaptive and learning organization is a challenging change process!

One issue to be addressed in this Chapter is how to succeed in creating a feasible system architecture in an agile environment. How can system architects, who usually act as multi-disciplinary mediators between organizational units and different stakeholders, leverage from agile principles and practices? And above all, how can model-based systems architecting also be used in a meaningful and helpful way in an agile project environment? The essential question is this: Does it fit together at all; a model-based system architecture design in an environment where every employee lives the agile values and principles?

In this Chapter, we try to give some answers to these questions. To do this, it is first important to make a short journey into the past and take a look on the historical origins of the agile movement and how it was influenced.

16.1 The History of Iterative–Incremental Approaches

There's a widely held view that iterative–incremental development is a relatively modern practice that has arised during the last two or three decades, but that's not true. Its historical roots dates back to the 1950s.

16.1.1 Project Mercury (NASA, 1958)

In 1958, the U.S. federal government agency National Aeronautics and Space Administration, better known under its abbreviation NASA, launched its first human spaceflight program: Project Mercury. The goal of this space program was to put a human being into orbit around the earth and also return it safely. Mercury laid the foundation for NASA's legendary Apollo program and was an important milestone in the so-called "Space Race," a popular term at the time for the competition between the U.S. and the Soviet Union during the Cold War for pioneering achievements and supremacy in space travel (Figure 16.1).

Project Mercury is especially also interesting from a different point of view: the approach chosen by the engineers for the development of Mercury's software! In a featured article about the history of Agile, published in the periodical "Computer" issued by the IEEE Computer Society in 2003 [22], we can read:

> Project Mercury ran with very short (half-day) iterations that were time boxed. The development team conducted a technical review of all changes, and, interestingly, applied the Extreme Programming practice of test-first development, planning, and writing tests before each micro-increment.

Figure 16.1 Mercury's Friendship 7 capsule at the National Air and Space Museum, Washington D.C, 2009. Source: HrAtsuo / Wikimedia Commons / Licence: CC0 1.0 (Public Domain).

This observation is particularly interesting in two respects:

On the one hand, the development of Mercury's software obviously proceeded iterative–incremental. Going forward in iterations means that they've chosen the way of an approximation to the exact or final solution with the help of similar steps of equal length. In this case, it is particularly noteworthy that these iterations (so-called "time boxes") were only half a day long, which is pretty short. Incremental means that the overall system to be developed is steadily increasing in its functionality, i.e. it moves closer to the final solution with each increment. One of the main reasons why it is strongly recommended to use an iterative–incremental approach in a complex environment is that it significantly improves predictability and minimizes risks.

Furthermore, they've also chosen a test-first approach. In other words, the developers approached the problem with having the acceptance criteria in mind, first defining test cases that the system under development must successfully meet. These test-first approaches became increasingly popular in the software domain in the context of the mentioned Extreme Programming (short: XP) methodology toward the end of the 1990s, especially through a widespread practice called Test-Driven Development (TDD).

Although, of course, no one was talking about "agile development" in the 1950s, from today's perspective one could say that in a ambitious systems engineering project from the space industry, such approaches were applied very early on. And as we know, they were markedly successful with it.

16.1.2 The New New Product Development Game (1986)

About three decades later, in 1986, an essay written by two Japanese communication scholars entitled "The New New Product Development Game" [241] was

published in the prestigious Harvard Business Review (HBR). Hirotaka Takeuchi and Ikujirō Nonaka have examined companies from Japan and the United States that were taking a new approach to first-class and innovative product development. Among these companies were Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox, and Hewlett-Packard. It is important to emphasize that the new way of working observed by the two scientists was not applied to pure software development, but to products such as copiers, a car (Honda), Personal Computers (PC), and two cameras!

Takeuchi and Nonaka observed that in those projects the members of self-organized teams passed the balls to each other as in the team sport of rugby and are working according to a "unique dynamic or rhythm." The whole HBR article uses metaphors from the rugby game to underline both the aspect of moving forward as a team and interacting with others in a "scrum"; the latter is a free kick after a rule violation in rugby. The HBR article also makes it clear that this new way of working emerged intuitively because people were free to develop and were not steered and controlled by management. The reward for this freedom were innovative products that not only surprised and convinced, but also represented quantum leaps in their domain.

16.1.3 Boehm's Spiral Model (1988)

Two years later, in 1988, the so-called Spiral Model was published by the American software engineer Barry W. Boehm [31]. Primarily targeted at software development, it proposed an alternative, risk-driven approach as a counter-design to the traditional and strictly sequential development processes Boehm called the waterfall model.

The waterfall model was first documented in 1970 by the American computer scientist Winston W. Royce (1929–1995) in his paper "Managing the Development of Large Software Systems." Royce [212] himself does not use the term "waterfall model," but he depicts the linear process model with the help of graphical representations that intuitively resemble a waterfall.

Basically, the waterfall model suggests that a software development project should to run in the same traditional and strictly sequential way as it is usual for the planning and construction of bridges and buildings:

1. First, an extensive requirements analysis had to be done until it was thought that all requirements had been elicited.
2. Afterward, an extensive architectural blueprint, design, and plan was created about how the software have to be "built" (so-called "Big Design Up Front").
3. Now it was hoped that a couple of programmers would just have to write down the source code off the cuff and implement the previously defined design.

4. Last but not least, they hoped that all they had to do was to test the software and they were done, usually ignoring the fact that the goal of a test is to find bugs which must be fixed.

It quickly turned out that complex software development does not work that way. In many software development projects where attempts were made to proceed in this manner, time and budget were often not met, or the stakeholder needs were not or only partially met, or the software was of poor quality. Interestingly – and often ignored by users! – already Royce described very vividly in his paper the high risks associated with this model, and that it is a very simple approach that explicitly works only well for very simple projects. In the same breath, he strongly recommended evolutionary models for complex development projects.

In general, it can be said that the development of a complex product – and this does not only refer to software – is a process that must be able to cope with changes, new insights, and surprises very well. And the process must support the possibility of being able to revise wrong decisions, i.e. learning experiences must be taken into account.

Barry W. Boehm's proposed Spiral Model envisions an iterative approach in which the various stages of a traditional approach are passed through multiple times, adding one stage to the product each time. Boehm himself called it an "evolutionary development model." It is better known as the Spiral Model, because it was usually visualized by depicting development iterations as a spiral around the center of a coordinate system like in Figure 16.2.

The four quadrants of the diagram correspond to the four main process steps of Boehm's evolutionary model. A central role plays the creation of prototypes (see upper right quadrant), which enable continuous evaluation and testing of the system to be developed. This serves to identify risks as early as possible and to be able to react to them appropriately. The continuous further development of the prototypes benefits from the increase in knowledge, so it is assumed that the prototype will grow continuously until the final, operational system is completed.

While the Spiral Model has obviously various advantages over a risky waterfall-like approach, there are also some drawbacks. For instance, its management is relatively complex and the effort seems to be unreasonable for small to medium size projects. There is also a risk that the spiral will not come to an end and that the end of the project cannot be accurately predicted.

16.1.4 Lean (1945 Onwards)

Other influencing trends came from the Japanese automotive industry, for example, such as Lean Management and Lean Production. Lean is an umbrella term for principles, methods, and practices for the efficient design of the entire

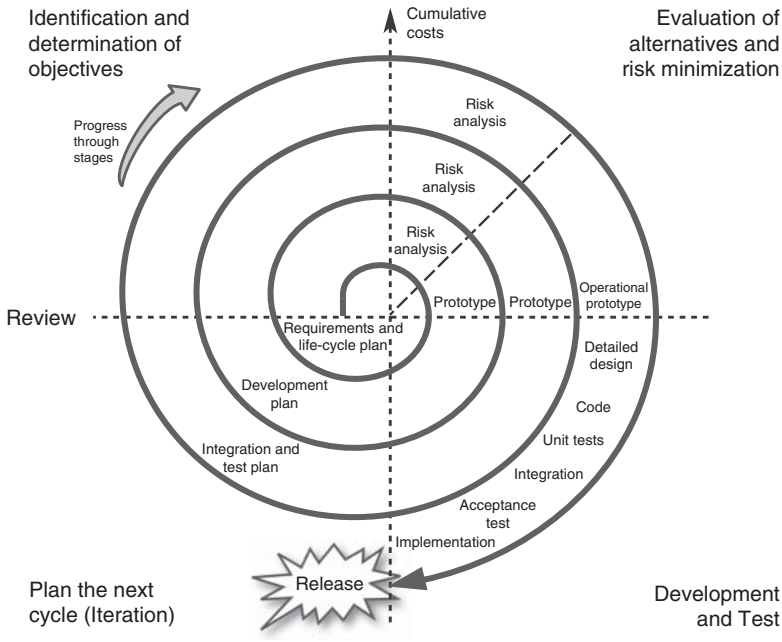


Figure 16.2 Adaptation and simplified representation of the Spiral Model according to Barry Boehm.

value chain of industrial goods. The global benchmark for Lean Production was and still is the Toyota Production System (TPS), which was established by the Japanese inventor Toyoda Sakichi (1867–1930) and systematically developed further after the Second World War by the engineer and production manager Taiichi Ōno (also: Ohno, 1912–1990) [192]. Ōno added many elements and methods to the TPS, e.g. the well-known practice called Kanban for controlling the production flow, which has also been widely used in other domains, like in software development. And another important concept in the context of Lean is Kaizen, a word which is Sino-Japanese and can be translated to "Change for better." Kaizen means that there must be a process of continuous improvement at all levels, involving all employees from the organization's CEO to the workpeople. It also aims to eliminate waste and redundancies, which are called Muda. For instance, in product engineering, it should be avoided to do things that do not add value or are plain superfluous. And another form of Muda is producing something that the customer neither asked for nor would pay for.

At INCOSE, there is a working group that focuses on lean approaches: The INCOSE Lean Systems Engineering Working Group. Some of their work results

have been summarized in Bohdan ("Bo") W. Oppenheim's book "Lean for Systems Engineering with Lean Enablers for Systems Engineering" [194].

16.1.5 Dynamic Systems Development Method (DSDM, 1994)

As discussed in Section 16.1.3, large parts of the software industry in the 1970s and 1980s tried to work mainly according to the so-called waterfall model. And as we know, this approach was often not successful.

In the early 1990s, a software development method became increasingly popular which was called Rapid Application Development (RAD). It was another response and counter-design to the plan-driven waterfall processes. In RAD, prototypes are created quickly and often with the help of special development tools, often used in addition to or sometimes even instead of design specifications. RAD is particularly well-suited to the development of software or other systems that are especially oriented toward (though not limited to) user interface requirements. At that time, many development tools emerged that were used to create graphical user interfaces (GUIs) for software applications very quickly and intuitive in WYSIWYG (What You See Is What You Get) editors.

However, it quickly became clear that RAD projects also needed a structure, a kind of approach, but this was initially lacking. Thus, the not-for-profit DSDM Consortium (Dynamic Systems Development Method) was born and assembled in 1994 in Great Britain. The goal of DSDM was to create a process framework that prioritizes a fast and efficient delivery of business solutions to clients, especially with the help of RAD approaches. In 1997, Jennifer Stapleton, who was the Technical Director of the DSDM Consortium at that time, published the first book on DSDM [234].

Since its invention, DSDM has evolved into a mature, comprehensive and iterative–incremental project planning and management methodology for software development. It is based on a well-formed philosophy and a set of principles. In particular, the DSDM principles had a significant impact on the Manifesto for Agile Software Development which will be discussed in the upcoming Section 16.2. Since 2014, the approach is called DSDM Agile Project Framework. In 2016, the DSDM Consortium refocused itself to primarily address the topic of Business Agility, which includes Agile in IT. With this refocus came also a rebrand to Agile Business Consortium.

One aspect of the DSDM Agile Project Framework is particularly noteworthy: Modeling is seen as one of its important key practices to support effective communication. The DSDM Agile Project Framework Handbook [243] even dedicates an entire chapter to the discipline of modeling.

16.1.6 Scrum (1995)

The Scrum metaphor from Takeuchi and Nonaka's HBR article survived: About 10 years later, in 1995, the two software developers Jeff Sutherland and Ken Schwaber presented SCRUM (at that time the term was still written in capital letters, which is rather unusual nowadays because it is not an abbreviation) at the OOPSLA conference in Austin (Texas). The title of the original conference paper was "SCRUM Development Process" [216]. Herein, Schwaber describes Scrum as a development process that assumes that the development of complex systems is an unpredictable and complicated process. Furthermore, the terminology changes in the course of the paper and Scrum is also described as a methodology. Interestingly, many well-known terms are missing that were added to Scrum later, for example Product Owner, Scrum Master, Retrospective, and a couple more.

In the following years, Scrum has been constantly evolved on the basis of experience. Nowadays, it is described as an empirics-based, iterative–incremental, and lightweight framework for product development. "Empirics-based" means that knowledge is generated from experience and decisions are made on the basis of observations.

It is particularly noteworthy that Scrum is nowadays defined as a "lightweight framework" rather than a development process or methodology. It is the nature of a framework that it is deliberately incomplete, i.e. users of it have usually to add something to make it work. In addition, the term "software" has been replaced by the more general term "product," which should emphasize that the Scrum framework can be used for product developments in the broadest sense, i.e. also in systems engineering. The main parts and the course of a Sprint in Scrum are depicted in Figure 16.3.

Scrum consists of only a few rules and defines five events (Sprint, Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective), three artifacts (Product Backlog, Sprint Backlog, and the Increment) and just three roles (Scrum Master, Product Owner, and the Scrum Team) that make up the core. As a framework, Scrum must be concretized and extended by techniques for the implementation of these events, artifacts, and roles in order to actually utilize it for a concrete project. The essential core of Scrum was separated from the implementation techniques in order to clearly define the central elements and impact mechanisms on the one hand, and on the other hand to let user's maximum freedom for individual adaptation.

Scrum is "team empowering," i.e. the Scrum Team manages itself and its members decide what to do and how to do it toward a common goal. Neither the Scrum Master nor the Product Owner are their superiors. Scrum is based on trust and relies on the collective intelligence of the people who use it to produce better results than when those people are given detailed instructions by managers.

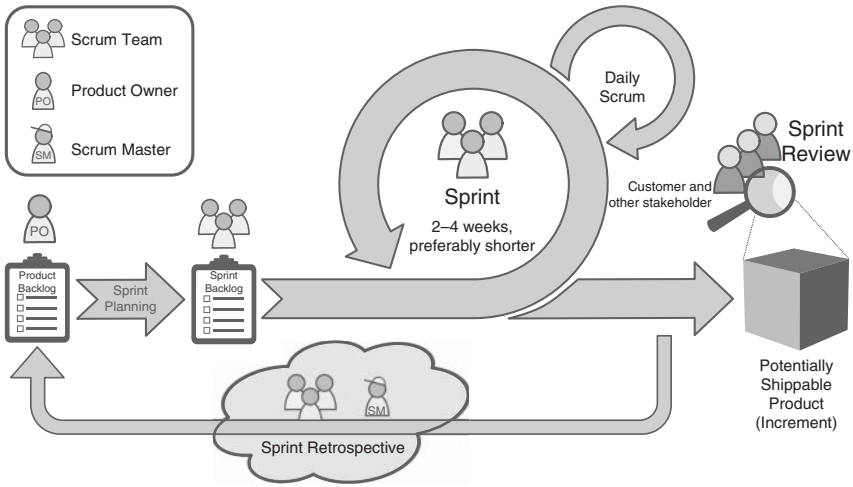


Figure 16.3 The development process according to the Scrum framework.

Although grown up in software development, Scrum has been adopted until today in many domains where essentially complex work has to be done. It is worth noting, that also Schwaber and Sutherland did not refer to their Scrum framework as an "agile process," "agile approach," or anything similar. Even in the famous Scrum Guide [238], which contains the definition of Scrum, you will look for the term "Agile" in vain. That Scrum and Agile are the same is unfortunately a widespread misconception! It is important to recognize that a development team can follow Scrum's rules and still violate many of the agile principles. You can simply use Scrum like any other tool. No doubt, Scrum, and Agile fit very well to each other and create powerful synergies. But you have always to keep in mind, that when you're doing one doesn't mean you can just assume you are also doing the other.

16.2 The Manifesto for Agile Software Development (2001)

In fact, the term Agile did not appear until the early 2000s. In February 2001, 17 well-known software engineers, among others also the Scrum inventors Ken Schwaber and Jeff Sutherland, and Arie van Bennekum who represented DSDM, met at The Lodge at Snowbird, a hotel in a ski resort in the Wasatch mountains of Utah. They called themselves the "organizational anarchists." They did not come together to create a new software development or project management methodology. Instead, they came together at this place to identify the lowest

common denominator of all those frameworks and methodologies that provide a counter-design to the heavyweight, inefficient, and document-driven processes that were widespread in the software industry in those days. The outcome of this legendary gathering was published on the Internet and has become known worldwide: The Manifesto for Agile Software Development [244].

The Manifesto for Agile Software Development (hereinafter also referred to only as the manifesto) is a public declaration that consists of four judgmental core statements, refined, and deepened with the help of 12 principles. For instance, it assigns a higher value to individuals and their interactions than to processes and tools. And as another example, the manifesto assigns a higher value to collaboration with the customer than the negotiation of contracts. Broadly speaking, it is an appeal for a cultural change and about how to succeed in developing and delivering software that satisfies the needs of a customer in an environment of ever-increasing complexity and uncertainty at all levels. And the 17 signatories of the manifesto agreed that this can only succeed with interdisciplinary, self-organized teams in a learning organization.

In the years that followed the publication of the manifesto, the agile hype set in. Many people and organizations joined the agile bandwagon. There were positive, but also not so positive developments. For example, the term Agile also became a kind of marketing label: Because it can be used as an adjective, it is very easy to put it in front of nouns: agile coach, agile tool, agile framework, …, agile whatever. The project managers cheered: "Look, a new project management method!," and the consultants were happy that there was something new to bill hours with. Agile became a complete industry. The intended culture change failed to materialize in many places, and a lot of teams merely used Scrum (… or what they thought Scrum is) just as another project management tool by ritualistically performing what is included in the framework. And some of the 17 signatories of the manifesto spoke up, noting that much of what is selled under the term Agile has nothing to do with what they had originally meant, for instance Dave Thomas in a blog post entitled "Agile is Dead (Long Live Agility)" [246].

On the other hand, there were also many success stories. Many organizations increasingly challenged their management structures and their corporate culture. In order to survive in a global market and to deal with increasing complexity, traditional management concepts from the industrial age were replaced by modern leadership. Especially in the software domain, but also in other industries, many self-organized teams managed to develop and deliver products with remarkably high quality quickly. Nowadays, it is the normal case in some domains for software that it is built, tested, and deployed to the production environment several times a day (Continuous Integration/Continuous Delivery).

In the meantime, a declaration has also been published under the title "The Foundation of Agile Systems Engineering" (https://www.agile-systems-engineering.com) that is a combination of two parts: the "Foundation for Complex Systems Engineering" complemented with a slightly modified ("tweaked") agile manifesto. One of its co-authors is, beside our author Tim Weilkiens, Arie van Bennekum, who was one of the signatories of the original manifesto. In the tweaked manifesto, the phrase "Working software over…" has been replaced by "Working solution over…" to emphasize that it can be applied to other domains than pure software development.

16.3 Agile Principles in Systems Engineering

Agile principles and product development frameworks such as Scrum also quickly attracted attention in Systems Engineering. But the concerns were high. Although iterative–incremental methods and frameworks have verifiable their roots in product development, there are still many reservations and doubts about using such practices in Systems Engineering. Developing a vehicle, an aircraft, a satellite or a farm machine in an "agile manner" – how is that supposed to work?

No doubt, of course there are certain differences to pure software development. If we look back at the genesis of the Manifesto for Agile Software Development (see Section 16.2), we can see that the original idea and intention behind it was to focus more on people and their collaboration. In addition, it can also be noted that many of these principles are not tightly coupled to software development. It is much more about a mindset and a new form of leadership. In the following sections, we would like to give you a few suggestions and ideas on how this can also be helpful in the discipline of systems engineering in general, and architecting systems in particular.

16.3.1 Facilitate Face-to-Face Communication

In the manifesto, we can read that business people and developers must work together daily throughout the project. And another agile principle from the manifesto states that the most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

In many systems engineering projects, however, it can be observed that the engineers and developers have little to no contact with the business stakeholders or the system's users. Sometimes, artificial barriers are even created by prohibiting the members of an engineering organization from clarifying open points and

other issues directly with those people. For instance, they create processes that makes it hard for engineers and external stakeholders to come together and to talk face-to-face to each other. All communication in both directions is then usually steered and handled by one to a few people. These "communication proxies" are not only bottlenecks. It is an organizational anti-pattern, because losses, misunderstandings, and a poor efficiency are unavoidable.

It is therefore important that direct communication, both between engineers and also between the development team and external stakeholders, is not only not hindered or even suppressed, but that a safe space is even explicitly created for it. Let people come together and talk to each other! That does not mean, that, for instance, a user of the system to be developed must be involved in every single technical detail. But it does mean they need to be close enough to the process to provide feedback and be available to answer questions from the development team. This gives the greatest chance that the final product will best satisfy the needs of the stakeholders.

16.3.2 Create a State of Confidence

One of the principles of the manifesto state, that working software is the primary measure of progress. The output of every so-called Sprint in Scrum, which is just another word for iteration, is called Potentially Shippable Product Increment (see Figure 16.3), sometimes also just referred to as a Potentially Shippable Product. It is therefore declared as "potentially shippable" because it is usually not intended to actually deliver it to customers. It is solely about being able to demonstrate for the customer and other stakeholders that the product has gained in business value and that it is of excellent quality. Furthermore, its purpose is that the development team receives qualified feedback. In other words, it serves to determine: Are we developing the right thing?

So, the goal of presenting the Potentially Shippable Product Increment to the customer is more about transparency and to build and keep a state of confidence between the development team and the stakeholders.

To be honest, it is hard to imagine that in systems engineering a Potentially Shippable Product Increment is created every three weeks if the system to be developed is, for instance, a satellite or a CT scanner. But if we think about the purpose of this outcome, we can find something comparable in systems engineering, especially in the early development phases, that could serve the same:

- A feasibility study to determine if the project should be pursued.
- A (possibly executable) SysML model of the system to be developed.
- A technology demonstrator, i.e. a test setup that evaluates the feasibility of a project.
- Using additive manufacturing (AM) technologies, like 3D/4D printing, for rapid prototyping of parts or even the whole system.

- An early prototype or design study of the system to be developed.
- Building a low-fidelity mock-up, e.g. out of cardboard/paper and other craft materials, to get early user feedback. (This is a technique from the creativity and innovation method Design Thinking.)
- Creating a digital twin of the system that can be explored interactively, for instance, with the help of Virtual Reality (VR) or Augmented Reality (AR) technologies.
- A subsystem or component of the system to be developed that is already functional.

16.3.3 Build Transdisciplinary and Self-Organized Teams

In their new definition, the International Council on Systems Engineering (INCOSE) describes systems engineering as, among other things, a "transdisciplinary approach" [227]. This is a notable change from the previous definition, which still referred to systems engineering as an "interdisciplinary approach." So, what is the difference?

Well, the term "interdisciplinary" can be regarded as an adjective that describes that something is related to more than one sphere of knowledge. In other words, two or more engineering disciplines, e.g. mechanical engineering, electrical engineering, and software engineering, are combined to a new level of integration.

This is also basically true for transdisciplinarity, but this goes even further. One speaks of transdisciplinarity when two or more disciplines transcend each other, i.e. they cross their boundaries, and thereby form a new holistic approach. The result will be completely different from what one would expect by just adding the parts; freely adapted from the Greek philosopher Aristotle: The whole is more than the sum of its parts. The point is that systems engineering shall dissolve domain-specific, disciplinary boundaries in order to solve problems in a way that is independent of specific fields and engineering disciplines.

Already in Scrum, the team was designed as a cross-functional, self-managing, and self-organized unit. And in the manifesto, we can read that the best architectures, requirements, and designs emerge from self-organizing teams. At that time, it was already clear to the authors that the artificially created silos in organizations culturally shaped by the industrial age are an impediment. And the manifesto further suggests to "provide motivated individuals with the environment and support they need and trust them to get the job done."

16.3.4 Create a Learning Organization

The term learning organization was first coined in the early 1990s by Peter Michael Senge, an American systems scientist who is a senior lecturer at the MIT Sloan

School of Management. In his book "The Fifth Discipline: The Art and Practice of the Learning Organization" [223], Senge described how the systems thinking method can be used to transform companies into learning organizations.

In a learning organization, the primary goal is not to install official programs for the acquisition of explicit knowledge. Rather, the interaction of culture, organizational design, leadership, and IT should create an environment that enable and promote informal learning, knowledge storage, and knowledge sharing. The following characteristics make up a learning organization:

- A learning organization has only few formal rules, hardly any regulations, and no rigid hierarchies. It is characterized by self-organization (see Section 16.3.3) and focuses on the acquisition, storage, and exchange of informal knowledge.
- The corporate culture of a learning organization is characterized by the fact that it promotes and demands the exchange of information and knowledge among employees and organizational units.
- A learning organization promotes the establishment of internal and external networks that serve to generate and exchange knowledge. Special IT systems are to provide support.
- Finally, the organizational structure, organizational culture, leadership, and technology/IT systems result in a systemic unity. With this systemic interaction, employees should be stimulated and motivated to learn as well as to share, store, and process their knowledge. In addition, the entire organization should join into an exchange of information and knowledge with its environment.

If we now compare the agile principles from the manifesto with these characteristics of learning organizations, there are strikingly many similarities. The Scrum framework also contains many things attributed to learning organizations, such as self-organized teams (see Section 16.3.3). In particular, the empirical generation of knowledge based on experience in order to be able to make better decisions based on this knowledge is also at the heart of a learning organization. To boil it down: Agile can be an enabler for a transformation into a learning organization.

16.3.5 Design, but No Big Design (Up-Front)

In many teams that adopt an iterative–incremental approach like Scrum and call themselves "agile," there is an opinion that they don't need architecture or at least don't need system architects. We are convinced that this is a very dangerous misconception! Both – agile and architecture – address the same challenge, that is, how to successfully develop a system under difficult conditions, such as ever-increasing complexity, permanent technological progress, and short-lived business models.

What certainly hardly works anymore is a so-called Big Design Up Front (BDUF), an approach in which the system's architecture is to be completed and perfected early in the project, as envisaged in the classic waterfall approach. However, it is certainly not wrong to create at least one, perhaps only relatively abstract, architectural design at the beginning of the project so that all project participants share the same product vision. This first architectural draft will be changed, extended, and refined in the same iterative–incremental manner as the system itself, based of course on empirical knowledge gained during the course of the project.

The essential point is this: BDUF is bad, but not creating an architecture at all is project hazardous! In contrast, a lean approach to the architecting of systems will blend perfectly well with agile principles and practices. The keys to success are "just-enough architecture" and "emergent design," i.e. the system's architecture should incrementally grow as the system itself.

Working with SysML models for the system architecture and design makes such an approach much easier. Implementing an MBSE approach provides teams the opportunity to respond quickly to change, analyze the impact of new or changing requirements, and ensuring a short period of time where teams can learn what specific architectural approaches work.

We would like to recommend the book "Agile Modeling" by Scott W. Ambler [18]. It addresses modelers that would like to improve their effectiveness during modeling by applying a similar focus on getting work done as developers who apply agile development practices. Even though the book primarily addresses software developers, its recommendations are suitable for modeling in systems architecting. It is full with hand-drawn sketches and white-board prototypes that remind us of the card technique for applying the FAS method in a workshop that we have described in Section 17.9.

16.3.6 Reduce Dependencies

Dependencies are basically bad. Unfortunately, dependencies are also unavoidable and inherent in the engineering of complex systems. Just one simple example: If a system element A is dependent on a system element B, and A and B are developed by different teams, then the team responsible for A is also dependent on the team responsible for B at the organizational level. That means: Especially in large development projects, you may end up with hundreds or even thousands of practitioners who are cross-dependent on each other in a variety of ways. The larger the number of dependencies, the larger the chances that a system's part will not be completed on time that may affect negatively other teams.

Especially when multiple teams are working on the same system, it is important to tailor the team structure so that teams are as independent as possible.

This is especially true if the teams work with an iterative–incremental approach like Scrum, and the output of one team in iteration n is the input for another team in iteration $n + 1$. Circular dependencies are an anti-pattern and should be avoided under all circumstances! It means that, for example, two teams are mutually dependent on each other, which leads to high coordination efforts. Moreover, such an interdependence on organizational level may also be an indicator for a problem in the system's architecture.

16.3.7 Foster a Positive Error Culture

An iterative–incremental approach like in Scrum also means that the result of an iteration sometimes does not meet the expectations of the stakeholders. When the product increment is presented to them, the feedback can be negative, or the product shows incorrect behavior during the demonstration and produces errors. This must not have dire consequences for the team members.

The English word for failure should be used positively: FAIL (First Attempt In Learning). Mistakes happen; they are part and parcel of learning and development. It is therefore important that an organization has a positive error culture and signals its acceptance of errors or failed attempts to its employees. An organization, on the other hand, that has a toxic culture and does not see mistakes as a learning experience, but instead looks for culprits and holds them accountable, is dead. People in such an organization constantly live in an atmosphere of fear and do not dare to break new ground, try things out and experiment. Innovations are thus made impossible.

In systems engineering, developing a new functionality can be very closely tied to exploring new ideas. And when you do research, you usually end up in unfamiliar territory, possibly doing something you've never done before. The key of success is to fail fast by trying multiple approaches and ideas and quickly evaluating their feasibility.

16.4 Scaling Agile

Scrum is designed for a single team with a recommended maximum size of up to 10 members. For a small business or start-up, this is usually enough, but what happens when a company grows? Furthermore, there are also large, traditional enterprises whose management decided to "become more agile," whatever the motivation for this decision may be. They are usually starting small as they look to leverage the benefits of Agile. They often begin with one single team and, for instance, introduce Scrum for a small pilot project. Starting small with something new is basically a good idea. And if it works and the project is successful,

management quickly wants to transfer the approach to other areas of the organization.

Scaling frameworks, which have been around for some time, promise to help. There are a couple of such scaled agile approaches available on the market, like:

- Nexus™, the approach by Ken Schwaber and scrum.org [217].
- Scrum@Scale™, created by Jeff Sutherland with Scrum Inc. [237].
- Disciplined Agile Delivery (DAD; IBM's process framework created by Scott W. Ambler and Mark Limes) [218].
- Scaled Agile Framework™ (SAFe™, created by Dean Leffingwell and Drew Jemilo) [131].
- Large Scale Scrum (LeSS, invented by Bas Vodde and Craig Larman) [260].

In particular, the latter two mentioned, SAFe™ and LeSS, are now very popular and quite widely used.

Discussing all these frameworks in detail is far beyond the scope of this book. For example, SAFe alone has in its current version named "SAFe 5 for Lean Enterprises" four configurations for different scaling levels: Essential SAFe, Large Solution SAFe, Portfolio SAFe, and Full SAFe. It is therefore already a challenge to first choose an appropriate scaling framework for an organization and their goals, because every case is different.

Apart from their differences, one thing that can be said about all of these frameworks is that they utilize Scrum at their core. In addition, some of these frameworks use the concept of an overarching "meta sprint," i.e. a longer-lasting sprint with multiple teams that in turn use shorter sprints. Basically, they combine well-known and proven concepts, such as Meta Scrum, Scrum of Scrums, Scrum, Kanban and the principles of Lean.

The application of such out-of-the-box blueprints to transform a classic organization into an "agile organization" seems appealing. The management of many organizations see those scaling frameworks as the saviors to finally become more Agile. And of course, you can also acquire a lot of certificates. But it is not that simple. It must be clear to everyone that the introduction of, e.g. SAFe is a very ambitious change project. Such a change can affect several hundred, or even thousands of people. A major challenge in such a transition are existing old behaviors, traditional role models, and hierarchical structures, which can very quickly become a showstopper. And just like you can utilize Scrum while disregarding all of the agile principles from the manifesto, you can also use one of these scaling frameworks just like any other multi project management tool. The desired transformation to an agile and learning organization then falls by the wayside: Neither leadership behavior nor corporate culture are changed.

The essential point is this: Some people treat these frameworks as a silver bullet. To be fair, this cannot be attributed especially to these scaling frameworks

alone, but is a prevalent misconception about Agile in general and Scrum in particular. But an agile and learning organization requires a cultural shift toward openness, trust, customer orientation, and cooperation. This cultural change cannot be imposed from above by management or brought about by a predefined scaling framework. Scaling frameworks may serve as inspiration, but there is no one-size-fits-all solution to this; a company must always find its own path toward becoming a learning organization.

16.5 System Architects in an Agile Environment

System architects are dependent on an involvement of stakeholders from different engineering disciplines in order to get architecture increments and other deliverables done. They are thus also dependent on the development approaches of their stakeholders.

As already discussed in this chapter's introduction, we see in many organizations that in different areas of engineering the agile values and principles are internalized and lived to different extent. For system architects, this sometimes results in a mixture of more or less agile and traditional approaches among the stakeholders and sometimes even in a mixture of different agile frameworks and techniques. Even if all stakeholders that are involved in a task are working according to the same iterative–incremental approach, e.g. Scrum, some practical issues may occur for system architects, for example different iteration length or non-synchronized starting points of the iterations in different organizational units. This results in some challenges for system architects working in an organization without a holistic agile approach, as described, for example, by Larmann and Vodde [155].

A system architect team should overcome these challenges by potentially adopting its own agile principles and practices, but also by being aware of and proactively coordinating the different working approaches of the various teams and stakeholders. Such coordination can be achieved, for example, through an organizational pattern that is well-known in the agile community as a guild. Agile guilds are the communities of practice for different competencies in engineering teams. A guild is a group of individuals who, while participating in various transdisciplinary engineering teams, meet regularly to share information and to discuss and resolve specific issues across teams. For example, a guild can be formed to address topics and issues about quality assurance and testing across teams. And as another example, there may be a system architecture guild instead of a fixed system architecture team. The difference is that an architecture team works together

permanently and its members primarily interacts with each other every day, while the members of an architecture guild sit on the various engineering teams and participate in achieving their goals while focusing on the system architecture and keeping the big picture in mind. It is obvious that a common system architecture model can serve very well as a communication aid here.

17

The FAS Method

The functional architecture is often mentioned in publications about systems engineering or in the context of real system projects. On the second view, you will realize that different terms are used for an architecture based on functions like logical architecture, logical view, or functional view. And different artifacts are part of those architectures: sets of functions, flow models, or functional models for simulation purposes. However, they all share the same notion: a technology-independent function-oriented description of the system.

Jesko Lamm and Tim Weilkiens have observed a lack of concrete common methods for functional architectures particularly in model-based systems engineering (MBSE). Some years ago, they described the Functional Architectures for Systems (FAS) method [153]. It was not a complete new method, but more a putting together of already existing puzzle pieces. The FAS method is a practice-proven method based on common MBSE practices.

We describe FAS in this chapter and start with our view on the terminology and motivation of functional architectures. Finally, we shine a light on different aspects of functional architectures like tool support, nonfunctional requirements, functional architectures for cyber-physical systems, and the role of technology in a technology-independent description. Parts of this chapter are based on our article "Method for Deriving Functional Architectures from Use Cases" [154]. We omit the citation of the article at each statement that is taken unchanged or updated from that article. This chapter is the official and most up-to-date description of the FAS method.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

17.1 Motivation

Functions are the essential core of a system. Their results are finally the reason why we build the system. They are the main important features, and any other aspect is secondary or depends on the functions. Therefore, it is crucial to make them explicit artifacts in the development process of a system.

The functional architecture enables the description of functions in an intuitive way that fits into the toolset of system architects. It is a description of systems independent of their technology in a block-oriented structure. System architects have their strength thinking in blocks and their relationships and not in functional flows, which is more the strength of requirements engineers.

An illustrative example of the separation of function and technology is photography. A function for taking photos was present already in the old-fashioned mechanical Camera Obscura. The Camera Obscura is no longer a state-of-the-art technology in photography, but modern cameras still need the related function of enabling the photographer to take photographs. Of course, the decomposed functions on the more detailed levels are different between the Camera Obscura and modern high-end cameras. These functions depend on the selected technology. Technologies introduce new functions that are directly bound to those technologies. See Section 17.11 about the relationship between technology and technology-independent functional architectures.

This example shows: describing products by their functions leads to concepts with a higher lifetime than an approach that depends on a certain technology. And the lifetime of technologies is steadily decreasing, and new technical components require an update of the functional deployment. Especially, a shift of a functional allocation from one engineering discipline to another, for example, from electrical to software, could be best analyzed in the holistic scope of the functional architecture.

We observe an increasing tendency that functions are deployed on many physical components and that a physical component provides more than one function (Figure 17.1). That advances the need for a clear understanding of the function structure. In the past, it was sufficient for an engineer to focus on the physical parts of the system. Simply speaking, one function was allocated to one physical part, and one physical part realized one function. The related functions of the physical parts were obvious and well covered. That is different if many functions are realized by the physical part, and a function is realized by many physical parts. Which are these functions? Which are the most important ones? What is really optimized if the engineer improves the physical part of the system? The functional architecture supports answering these questions.

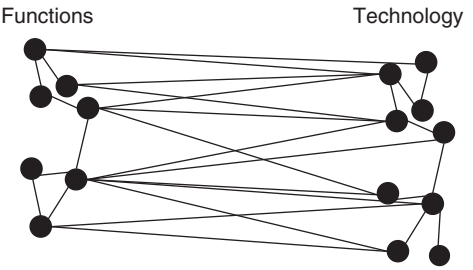


Figure 17.1 Functional-to-physical mapping.

A functional view also enables a deeper insight into the system [5, 103]. By using functional architecture descriptions, system architects can model their understanding of the system-of-interest without the complication of simultaneously designing a technical solution. Functional architectures are a well established concept (for example, [25, 38, 103]).

There are gaps between requirements and physical architectures that make it difficult for projects to succeed. An organizational gap: Requirements engineering and system architecture are different disciplines with different roles and usually different people behind the roles. Quite often, the people are organized in different departments and work at different locations. A cultural gap: Requirements engineers and system architects often have different educational backgrounds and mindsets. A technical gap: Requirements and architectures are often stored in different tools with no or bumpy connections. Although you can model a mapping relationship between the architecture and the requirements, there is a gap, and the other side gets easily out of the view from the perspective of the requirements or architecture.

These gaps lead to misunderstandings in communication, conflicts, errors, and so forth, and finally to missed deadlines, high costs, and dissatisfied stakeholders.

The functional architecture does not completely fill the gaps but is a good bridge pier to narrow the gaps. It is closely related to the requirements and, at the same time, part of the convenient environment of the system architect's toolset in the form of a block-oriented view. It is a place where both disciplines meet and get connected.

According to the INCOSE Systems Engineering Handbook, the functional architecture is the foundation for the system architecture through the allocation of the functions to system parts [265]. The systems engineering manual of the Federal Aviation Administration describes the functional analysis as an activity "significantly improves design, innovation, requirements development, and integration." [180].

Since the functional architecture is an essential and comprehensive description of the system functions, it is also a suitable artifact to be used for functional

safety assessment. One typical step of a safety assessment is to identify the appropriate functions, for example, ISO 13849-1:2006 [118], that can be found in the functional architecture.

17.2 Functional Architectures for Systems

Model-based functional architecture descriptions model systems independent of their target technology by means of functional elements that transform modeled information (signals, data), materials, force, or energy [199, 253].[1] The functional elements have different names in the literature, for example, functional element [277], function [140], or system block [29]. Hence, it is necessary to define terms that relate to functional views of the system like we use them in this book. This has been done in the glossary in Appendix C. Definitions of more common architecting terms such as architecture or architecture description are listed in Chapters 5 and 8. In addition to the definitions in the glossary in Appendix C, Figure 17.2 shows the functional architecture terms defined in a domain model with further information.

The functional architecture is based on functional elements, functional interfaces, and architecture decisions. It includes a hierarchical function structure that results from functional decomposition. The concept of functional decomposition can be found in a variety of sources ([197]: 170; [66]: 145–146; [25, 232]; [199]: 66; [199]: 199]). These sources differ in background, and not all have a systems engineering context. We interpret functional decomposition for systems in general. We thus do not specifically emphasize one field of engineering, as it is done, for example, in [197], where there is a strong focus on mechanical design.

A functional element encapsulates functions and defines functional interfaces to enable functional object flows to other functional elements defined by functional connections. The function represents a grouping of use case activities which is the bridge between the functional architecture and the requirements analysis. The functional interface defines a set of input and output parameters of a functional element.

The functional architecture is similarly defined in the Systems Engineering Body of Knowledge as "a set of functions, and their sub-functions that defines the transformations of input flows into output flows performed by the system to achieve its mission" [220]. This definition conforms to our definition of the functional architecture. The functional architecture does not consider the execution order

1 Force will sometimes be omitted in the discussion of such flows, as it is the derivative of energy.

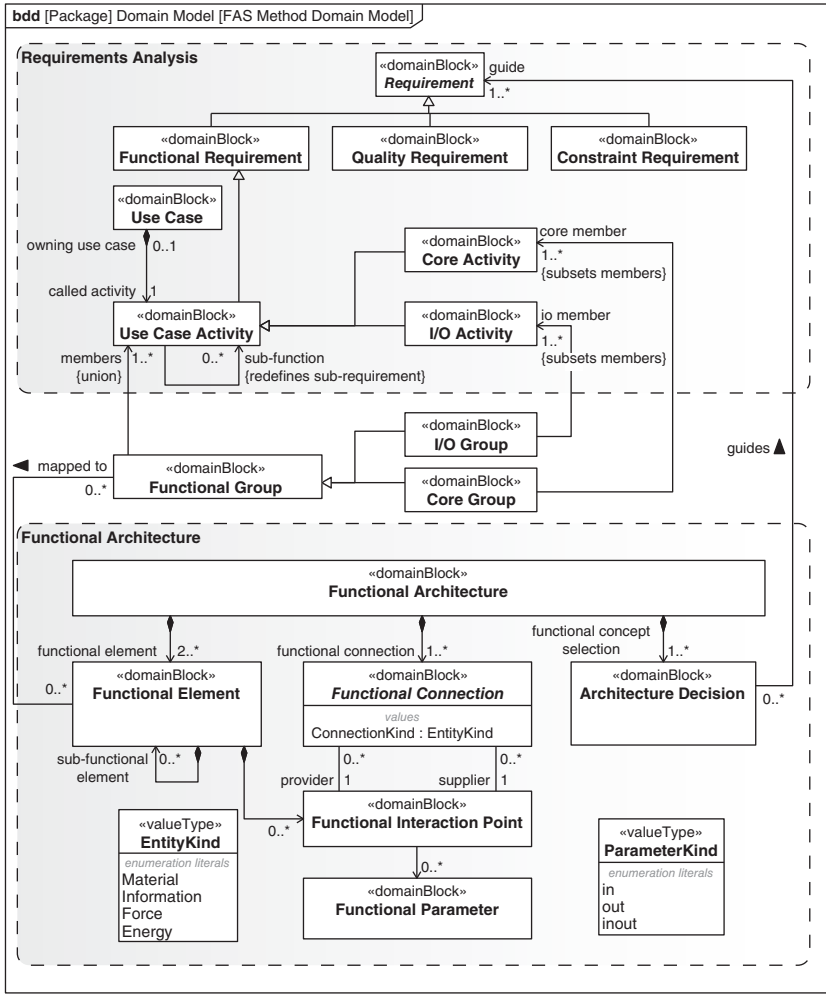


Figure 17.2 Domain model for functional architectures.

of functions but only the transformation aspect. Execution orders of functions are covered by behavioral architectures. However, we have also seen definitions that add the execution order into the scope of functional architectures.

The functional architecture is derived from the functions identified in the requirements and use case analysis (see Chapter 10). The functional grouping is the key for establishing a mapping between the functional architecture and the requirements. The functional groups include use case activities to derive the functional elements as fundamental bricks of the functional architecture.

Table 17.1 Mapping of FAS concepts to SysML

FAS concept	SysML element
Use case activity	Activity defining use case behavior
I/O activity	Activity whose call behavior actions are part of an activity partition with stereotype I/O
Core activity	Activity whose call behavior actions are part of an activity partition without stereotype I/O
Functional element	Part property typed by a functional block (block with FAS stereotype "functional block")
Functional group	Block with FAS stereotype "functional group"
Functional interaction point	Proxy port typed by an interface block
Functional parameter	Flow property of interface blocks that type functional interfaces
Functional connection	Connector between proxy ports of functional blocks
Functional interface	Functional connection including the connected functional interaction points, that means SysML connector and connected proxy ports

A good practice is the separation of functions responsible for interface functionality at the system border and the other functions (see also Section 10.2.4). The FAS method introduces the I/O group and the core group as special functional groups to separate the appropriate functions.

The functional architecture is a concept independent of SysML. Table 17.1 depicts the mapping of the FAS concepts to SysML model elements. The mentioned stereotypes are part of the FAS profile, extending SysML by the FAS concepts.

The functional interaction point specifies the functional input and output parameters of a functional element. The grouping of the parameters to functional interaction points is done by the system architect based on criteria that are relevant for the architecture. The functional connections connect the functional interaction points linking the outputs with inputs. The set of a functional connection together with the connected functional interaction points represents a functional interface.

How to model functional architectures with SysML is described in Section 17.5. You can replace SysML with other notations, for example, text written in natural language, spreadsheet documents, or your proprietary graphical notation.

Section 17.3 describes how to specify a functional architecture with cards and whiteboards in a workshop.

Functional architectures are independent of SysML, while they fit together perfectly. Next, we give a SysML independent description of the FAS method, although we use SysML to visualize the concepts of FAS.

17.3 How the FAS Method Works

Jesko Lamm and Tim Weilkiens described a method for deriving a functional architecture description directly from use cases in an intuitive and traceable way [154]. They called it the FAS method (functional architecture for systems). The tasks are performed by the system architects in collaboration with the requirements engineers using a clear interface and handover between both roles. The requirements and use cases are represented mainly in text and flow diagrams that are well suited for requirements engineers. In contrast, the functional architecture of the FAS method is a structural description in a block-oriented representation that is well suited for system architects.

The FAS method was first published in a paper for the German systems engineering conference TdSE in 2010 [153]. The first international publication of the FAS method was the paper "Method for Deriving Functional Architectures from Use Cases" in the Systems Engineering Journal [154]. This chapter is an extended version of that paper. Since its initial publication, the method has been used in various industrial and research projects (see Section 17.8).

The FAS method derives functional architectures in a block-oriented form from use cases via the grouping of use case activities. The method is based on standard techniques in systems engineering, like the identification of the system context and subsequent use case analysis (for example, [267, 271]). A brief description of these tasks can be found in Sections 8 and 10.2.3. They are not part of the FAS method but deliver work products as input for the method.

In Section 10.2.3, you find some results of the use case analysis of the virtual tour example system. Use case activities are a grouping of functions according to use cases, which means the grouping criterion is the usage of the system from the actor's perspective. The functional grouping of the FAS method applies a different grouping criterion to them. The criterion is the cohesion of the use case activities, which means use case activities that belong to the same subject are in the same group. While the use case view is important for the requirements analysis and to involve the domain stakeholders of the system, the functional view is important with respect to the system architecture to implement the functionality satisfying the requirements.

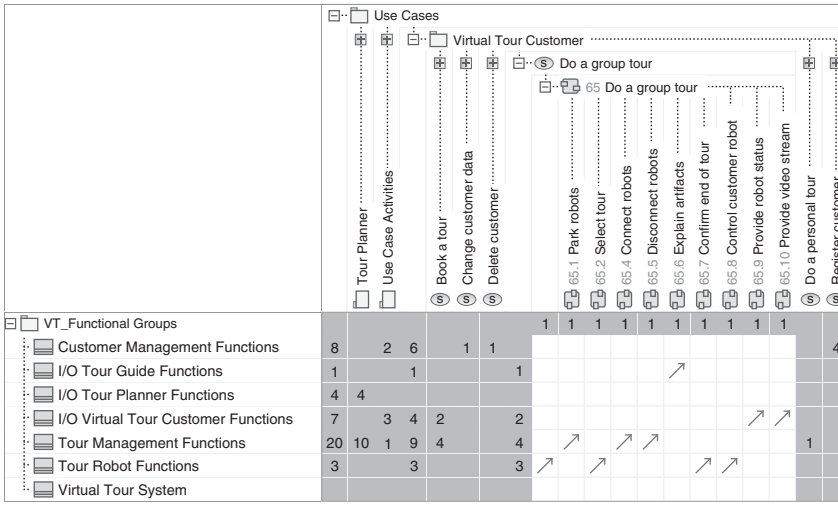


Figure 17.3 Matrix functional grouping.

A key step of the FAS method is the switch of the grouping of use case activities from use case oriented grouping into functional groups. The grouping allows keeping related aspects of the system's functionality closely together. This avoids, for example, developing different physical system assemblies for the same functionality. This part of the FAS method is the one that most requires the expertise of the architect and can also be supported with heuristics presented in Section 17.4.

Figure 17.3 shows an example of a functional grouping. We put all use case activities in a matrix where the activities are the columns, and the functional groups are the rows. That matrix gives a good overview of the grouping and is a handy tool.

Use case activities related to external interfaces are grouped in special "I/O" groups. That follows the heuristic "A functional group takes the functions that are related to a system actor"described in Section 17.4. System architects could then put his focus on the non-I/O functions that are, typically, more essential for the system.

Top-level root use case activities are sometimes too comprehensive to be completely assigned to one of the functional groups. In that case, they are assigned to a special top-level functional group with the name of the system-of-interest.

A functional group can contain other functional groups, and each functional group except the root group is a member of exactly one other functional group (tree structure). The best practice for your project depends on many criteria like team organization, focus on usability requirements, and more. Finally, the structure of the functions is a decision of the system architect.

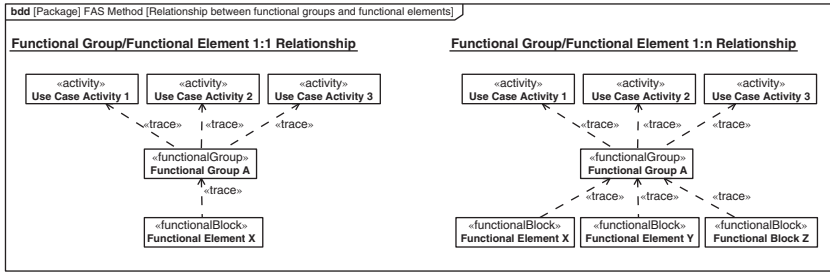


Figure 17.4 Relationship between functional groups and functional elements.

Initially, functional groups are mapped to functional elements of the same name. The transition from functional groups to functional elements decouples the life cycle of use cases from the one of the functional architecture, facilitating the implementation of change control across systems analysis and system architecture. For the first step, each functional group is mapped to exactly one functional element. Later the functional groups could be different than the functional elements. For example, the system architect refines a functional element and splits it up to three functional elements. All three functional elements relate to the same functional group (Figure 17.4).

As long as there is a 1 : 1 relationship between functional groups and functional elements, there is a clear traceability from the functional element to the functional group and to the use case activities. In the left part of Figure 17.4, it is clearly specified that the functional element "Functional Element X" represents the functionality of the three use case activities. If, for example, three functional elements relate to one functional group, it is no longer clear which use case activities are represented by each functional element (right part of Figure 17.4). If you would like to keep the traceability, you must keep the functional groups and the functional elements in a 1 : 1 relationship. The price for traceability is the loss of the decoupling as described above. If functional groups and functional elements are always in a 1 : 1 relationship, they can be implemented by the same model element.

Functional elements and their functional connections can be modeled in block-oriented form, for example, with SysML, as illustrated in Figure 17.5. The connection specifies that the output of functions from one functional element is an input of functions of the appropriate connected functional element.

The connections are linked via functional interfaces with the functional elements. The functional interfaces describe the input and output parameters of the functional element.

A functional element could have more than one functional interface to group functional parameters according to aspects that are important for the system architect. The parameters of a functional element can be derived from the inputs and

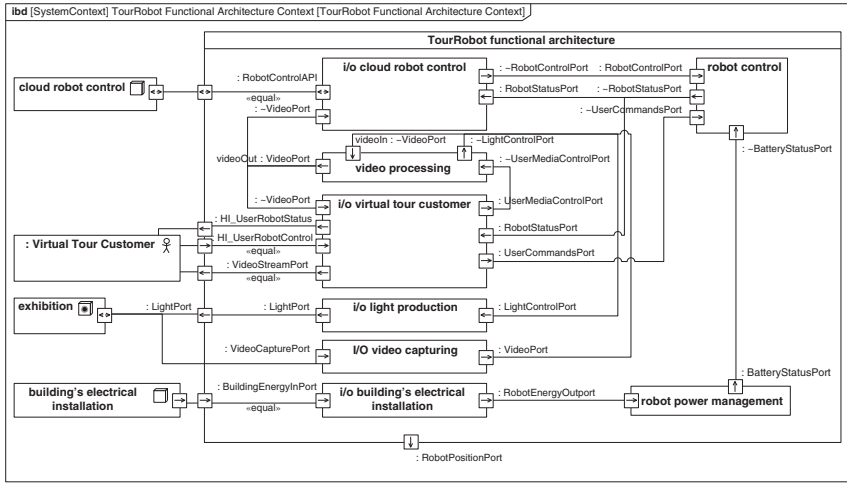


Figure 17.5 Part of the functional architecture of the TourBot.

outputs of the appropriate use case activities. The functional elements, together with functional connections and architecture decisions, are the functional architecture. Figure 17.5 depicts an extract of the functional architecture of the tour robot, which is part of the Virtual Tour system. Section 17.5 describes in more detail how to use FAS with SysML.

17.4 FAS Heuristics

Architecture combines art and technology and cannot be automatically created. Nevertheless, some automation can support system architects. The main work is based on the experience, creativity, and competency of the architect and is guided by architecting principles (Chapter 9). In addition to automation also heuristics can support the architect [164]. In the following, we provide heuristics to create groups of use case activities. The grouping aims at functional groups that are "as independent as possible; that is […] with low external complexity and high internal complexity" ([164]: 28). This is a known concept in classical architecture [13] and has been discussed by Baylin [25] regarding the functional modeling of systems. According to Baylin, functions should be clustered according to "functional cohesion" ([25]: 32), which means with the aim of keeping elements with related or same objectives together. See also Section 9.3 about the cohesion and coupling principle. For supporting the architect, Lamm et al. [151, 154] have found heuristics that aim at functional cohesion.

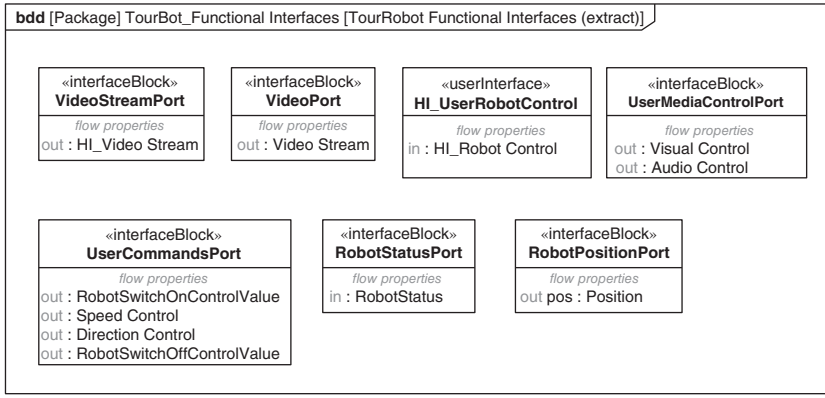


Figure 17.6 Definitions of some interface blocks.

A functional group takes the functions that are related to a system actor: Functions having a direct relationship with system actors are part of the system's input/output logic. Often they only have little in common with the actual system functions that do the processing of the inputs and produce the outputs and are a cohesive set of functions on its own. In that case, they are good candidates for a separate functional group whereby each actor gets its own functional group.

Function calls imply cohesion: Functions call other functions that often relate to a similar topic, resulting in a network of call relationships. Clusters in that network are potential functional groups.

Functions that share data can be grouped: It can be assumed that two functions belong to closely related domains if the kind of output of one of them is the other's input. This is more applicable if the kind of object is not a common entity type like length or speed but special and related to the domain. This connection can easily be found in the object flow of the use case activities. The so-called activity trees of SysML can facilitate the assessment of common data (Figure 17.7).

Use grouping criteria of existing groups: A system is rarely developed completely from scratch, but it is usually based on an existing system. The outline of existing system documentation from prior or similar systems can indicate possible ways of grouping functions. Ideally, interviews with the system developers should be made to find out if the grouping was useful in practice. This way, known structures will be created, and team members will find them intuitive to use. However, grouping criteria have to be re-assessed with caution: they can be of a technical rather than conceptual nature. A grouping based on technical constraints is not desirable because it will lead to a functional architecture that

contains implicit technological decisions, making it more difficult to find alternative solution scenarios.

Reduce the number of functional groups that include functional variation points: A variation point marks an element in the model that could be realized in different variants of the system. See Chapter 18 for more details about the modeling of variants. Reducing the number of functional groups that include functional variation points means to group those functions in as few as possible functional groups. The heuristic follows the architecture principle to separate stable from unstable parts (Section 9.5). More concrete G. Schuh states that variant full assemblies should be separated from assemblies with low variability [215].

Remember that all these are heuristics and not rules. They can support and guide the system architect but not take over the decision for a functional group.

17.5 FAS with SysML

While the FAS method is independent of any language or tool, we will show its application using SysML [189] because this language is well suited for implementing it. We recommend using SysML for the following reasons:

- SysML is an international standard. It is well-known, mature, and fits in most collaborative tool landscapes.
- SysML provides all necessary model elements for the FAS method.
- SysML is also suitable for the adjacent models of a functional architecture like requirements and the physical (logical and product) architecture models.

We've previously introduced the FAS method already with the SysML notation. Now we'll have a deeper look on the SysML model elements used for the FAS method.

17.5.1 Identifying Functional Groups

The functional architecture, as we introduce it here, belongs to the structural view of the system and does not include system behavior. As a consequence, the control flows of use case activities are less important in the FAS method, whereas the object flows are of major importance. The object flows connect the inputs and outputs of use case activities.

SysML provides a structural view of activities that hides the control flow. It is the activity tree (also called "function tree") in the literature (for example, Weilkiens [267, 269]) depicted in a block definition diagram, in which each node represents

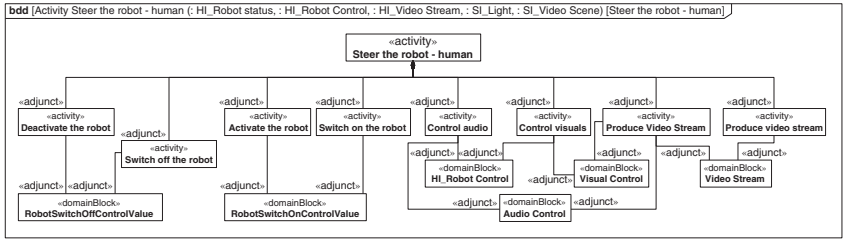


Figure 17.7 Example: Parts of the Activity Tree of the use case "Steer the robot – human."

an activity. Figure 17.7 shows an activity tree for the use case activity "Steer the robot – human."

The tree structure expresses the functions' call hierarchy; that means: a node calls its child nodes. This does not make the activity tree a functional decomposition where the node owns its child nodes. The lines connecting the activities in the tree are the SysML part association relationships. The properties typed by an activity are SysML adjunct properties. They constrain the values of the properties to be determined from the appropriate call behavior action. The tree also shows the input and output parameters of the activities, which are typed by domain blocks. Again, they are modeled as SysML adjunct properties. The property owns a link to the appropriate activity parameter called the principal.

The roots of activity trees are the activities that have a one-to-one relationship to the use cases. Each use case owns exactly one root use case activity that specifies the overall behavior of the use case. These activities have the same name as the use cases they relate to, as an exception to the naming convention for activities that the root activities are named from the perspective of the system actors and not from the system perspective like all other activities that are based on system functions.

The nodes of the activity trees are SysML activities and not SysML actions. In SysML, these are different concepts [267, 269]. The SysML activity is the whole behavior depicted by an activity diagram, and the SysML action is an atomic part of an activity. Functional decomposition is modeled with call behavior actions that call an activity which acts like a decomposition. The part properties at the end of the association relationship represent the call behavior actions. They are SysML adjunct properties that have a direct link to the appropriate action.

It has to be noted here that even though the activity tree view, as shown in Figure 17.7 is suitable to identify functional groups, and although SysML provides the tree view, it is not well supported by modeling tools due to several reasons. In addition, the trees may grow very large and difficult to handle. Therefore, it is only partly suitable in practice but a useful illustrative view for understanding the

source of the functional architecture. A handy replacement of the view to elaborate the functional architecture is the matrix view of relationships that is provided by most SysML modeling tools. The columns represent the activities and eventually actions, the rows the functional groups, and the marker in the cells represent the grouping relationship (Figure 17.3). The functional groups include activities and actions.

Despite of the functional architecture, it is best practice in use case modeling to model each use case step with an activity to define it and a call behavior action to describe its usage. It is another example of the definition/usage/runtime pattern (Section 9.4). If you follow this principle, all the system functions are specified by activities.

If you need other action kinds than the call behavior action, for example, for simulation purposes, be aware that these actions are not considered by the activity trees with SysML as we describe it here. However, you can tailor the usage of SysML for FAS to include any kind of action. In that case, you cannot use the tool of activity trees to identify functional groups. But they are just a tool and not mandatory for the FAS method. Instead of activity trees, you can use matrices to visualize and create the relationship between the activities or actions and the functional groups. A functional group can include activities as well as actions and functional requirements that are not refined by use cases. It could be any element that represents a function definition.

However, be careful not to mix up activities and actions. A call behavior action is a usage of an activity and not a definition of a function. In that case, the called activity is an element of the functional group and not the call behavior action. Same for the call operation action that is a usage of a block operation. It is the operation that defines the function and is part of the group. An opaque action is a definition and usage of a function at the same time and could be an element of a functional group. Actions to receive or send signals are also potential candidates to be a member of a functional group.

In SysML, the functional group is represented by a SysML block with the stereotype «functionalGroup» from the FAS profile.

The membership relationship between a functional group and the activities is the SysML trace relationship. The functional group is the source of the trace relationship.

17.5.2 Modeling the Function Structure

In SysML, the function structure is represented in block definition diagrams. Per functional group, one functional element is modeled as a SysML block with the stereotype «functionalBlock» defined in the FAS profile. To keep the traceability

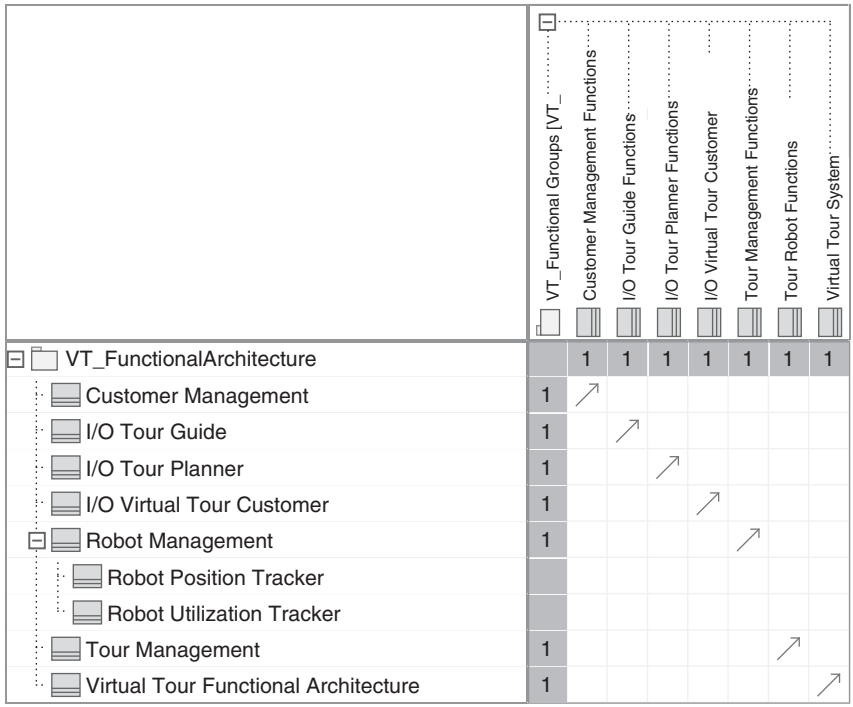


Figure 17.8 Traceability between functional blocks and functional groups.

between the functional block and the functional group, a trace relationship is modeled from the functional block to the functional group.

Again, that relationship could be best shown and managed in a matrix. Figure 17.8 shows the trace relationship between the functional block and functional groups of the Virtual Tour system. At a glance, you can spot that the functional block "Robot Position Tracker" and "Robot Utilization Tracker" have no trace relationships to functional groups. They were added later by the system architect to refine the functional block "Robot Management." There is no appropriate functional group, and the traceability to the use case activities and requirements is established by the enclosing functional block "Robot Management" that has a trace relationship to a functional group.

The operations of the functional blocks could explicitly model the actual function. As a rule of thumb, each operation of a functional block should match a function of the corresponding functional group. The functions of a functional group are, typically, the use case activities. You can establish the traceability from the operation to the use case activity by sharing the same name or, more formally,

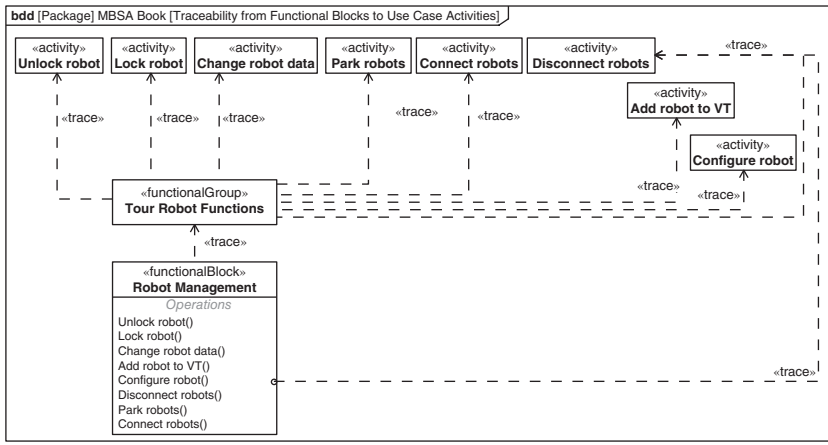


Figure 17.9 Traceability from functional blocks to use case activities.

but with more effort with a trace relationship from the operation to the activity element. The latter could again be managed with a relationship matrix.

Note that with modeling this explicit trace relationship or implicit relationship via name similarity, you lose the decoupling of the functional architecture from the use case model that you get by the indirect relationship via the functional groups. Figure 17.9 depicts the relationship between the operation "Disconnect robots()" and the appropriate activity. The topology of the diagram clearly shows that the decoupling of the functional group is bypassed by the trace relationship between the operation and the activity.

If you do not need that level of detail in the functional architecture, you can skip the modeling of the operations, which is the usual case. If you need that level of detail, you can connect the operations of the functional blocks directly with the appropriate activities and discard the modeling of functional groups. In such a scenario, the functional groups are just a tool to support the derivation of the functional architecture.

Functional blocks can be described as parts of other functional blocks using the composite relationship. This can be used to model the decomposition of functional elements into sub-elements.

Figure 17.10 depicts an extract of the functional architecture of the tour robot, which is part of the Virtual Tour system, and Figure 17.6 some of the interface block definitions of the used ports. The root element is a special functional block that represents the complete functional architecture. It has the stereotypes «functionalBlock»and «system»applied.

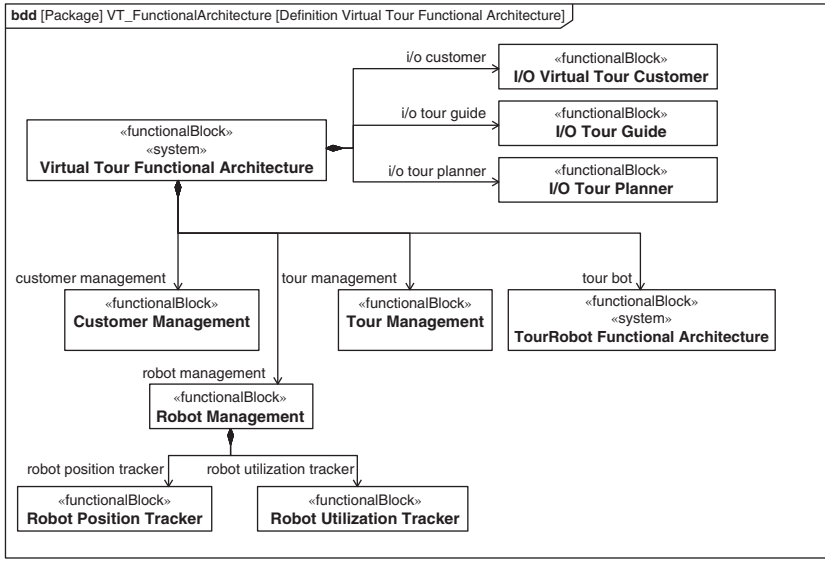


Figure 17.10 Function structure of the virtual tour system.

17.5.3 Modeling the Functional Architecture

The hierarchical function decomposition does not define how the functional elements are connected. The internal block structure depicted by the internal block diagram defines the connections between the functional elements. Part properties typed by functional blocks whose functions output is an input of functions of other functional blocks are connected in the internal block diagram using proxy ports with flow properties for the input and output data. If the proxy port does not delegate the flows to or from the internal parts of the functional block, it should be marked as a behavioral port, which defines that the owning block of the port implements the port.

Figure 17.11 shows one possible functional architecture description of the Virtual Tour system in a SysML internal block diagram. The part properties depict only the name of the property and not the type. Here, the name is identical to the type name but written in lower case. The functional architecture is shown within its context, which allows us to also display the external interfaces and the actors it is connected with. Hence, the interrelation between system functions and external systems or system users can be visualized.

Figure 17.12 shows the definition of the port types of the functional blocks. The port types are not yet complete. The *HI_CustomerPort* is a user interface.

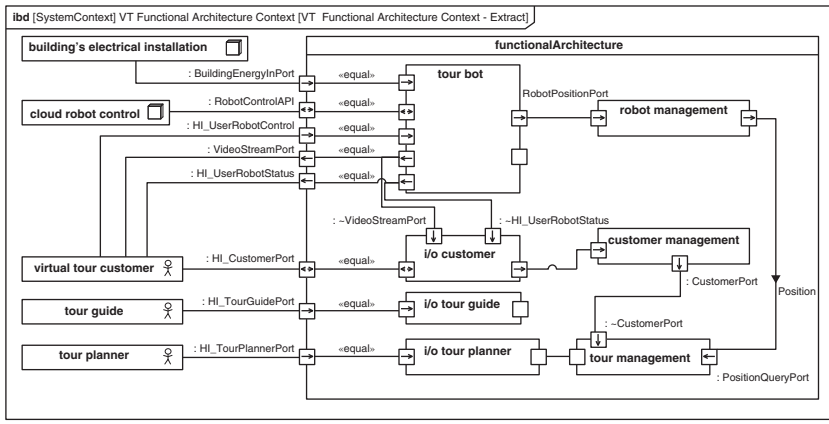


Figure 17.11 Example functional architecture of the virtual tour system.

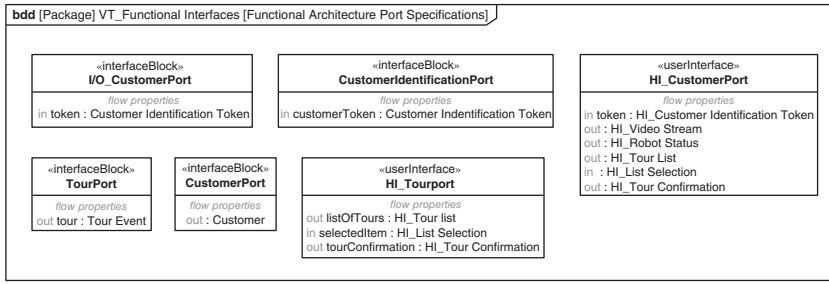


Figure 17.12 Types of the functional block ports.

«userInterface» is a stereotype of the SYSMOD profile to mark port types that describes the direct interface to a human user. Another port at the functional block "i/o customer" specifies the interface to handover the data to the customer management functions.

The functional architecture could be assessed using assessment methods described in Chapter 21 and architecture principles described in Chapter 9 like the cohesion and coupling principle.

17.6 SysML Modeling Tool Support

The FAS method interlinks use case analysis with the creation of a functional architecture. This enables traceability between requirements, use cases, and functional elements if there is a modeling repository or tool chain that allows

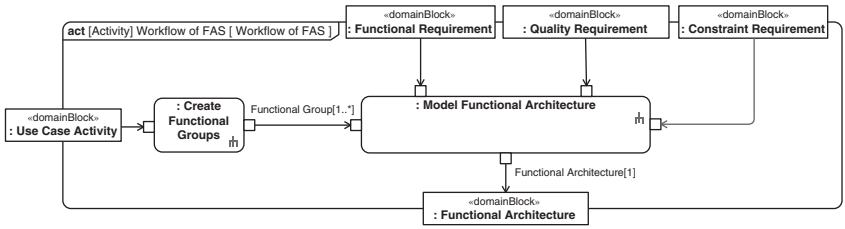


Figure 17.13 FAS process.

for linking both. Once a modeling tool has been chosen, it is usually possible to script certain modeling steps in order to automate the user's work and thus make modeling-intense work less error-prone. Indeed, the following parts of the FAS process from Figure 17.13 are well suited for being automated [144, 154]:

- The creation of initial functional groups in the step "Identify Functional Groups."
- The creation of initial functional blocks and their interfaces in the step "Model Functional Architecture."

By the time of writing this book, the open-source community supports the FAS method with add-ons that are offered for download for different modeling tools (www.fas-method.org), providing some of the possible automation.

In the following, one typical approach of the automation will be described. For simplicity, the term FAS engine will be used for the modeling tool add-on that supports the FAS method. To illustrate the split of the FAS method's steps between manual work in the SysML editor and automated actions of the FAS engine, Figure 17.14 shows the detailing of the task "Create Functional Groups," and Figure 17.15 shows the detailing of the task "Model Functional Architecture" from Figure 17.13. The split between manual and automatic tasks is modeled by partitions.

17.6.1 Create Initial Functional Groups

The heuristic "A functional group takes the functions that are related to a system actor" can be supported already during the use case analysis phase: the call behavior actions of the use case activities are split into activity partitions representing core functionality of the system-of-interest and those relevant for interfacing it to its actors. The activity partition of the latter kind can be marked with the stereotype «I/O» from the FAS profile [144].

Figure 17.16 shows an example of the use case activity "Book a tour" with activity partitions. On the first view, there seems to be a mistake in Figure 17.16. The input parameter "HI_Customer Identification Token" from an actor flows directly

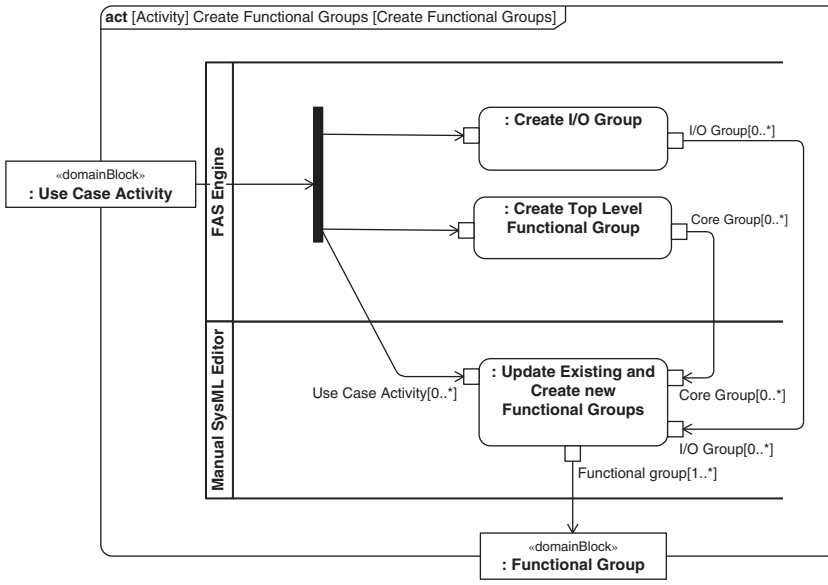


Figure 17.14 Manual and automated steps of the FAS task "Identify Functional Groups."

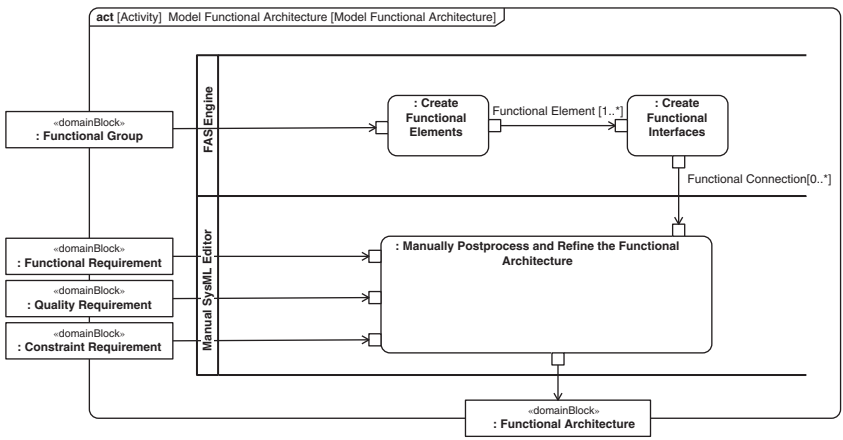


Figure 17.15 Manual and automated steps of the FAS task "Model Functional Architecture."

in an action of the core partition and not in an action of the I/O partition. On the second view, you will see that the activity "Identify customer" has its own I/O partition (Figure 17.17). There the input parameter "HI_Customer Identification Token" is an input of an action of the I/O partition.

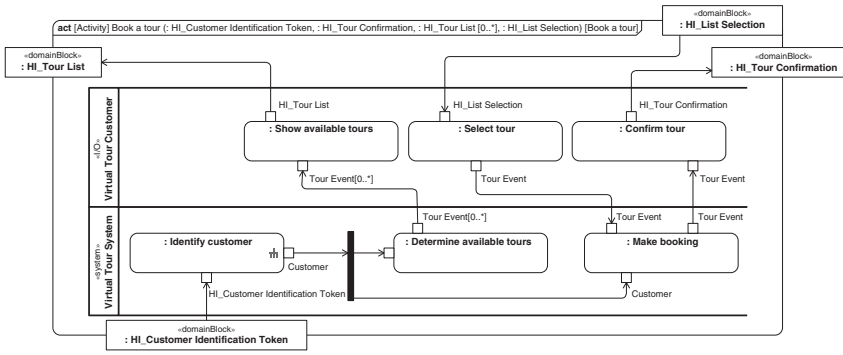


Figure 17.16 Example use case activity "Book a Tour."

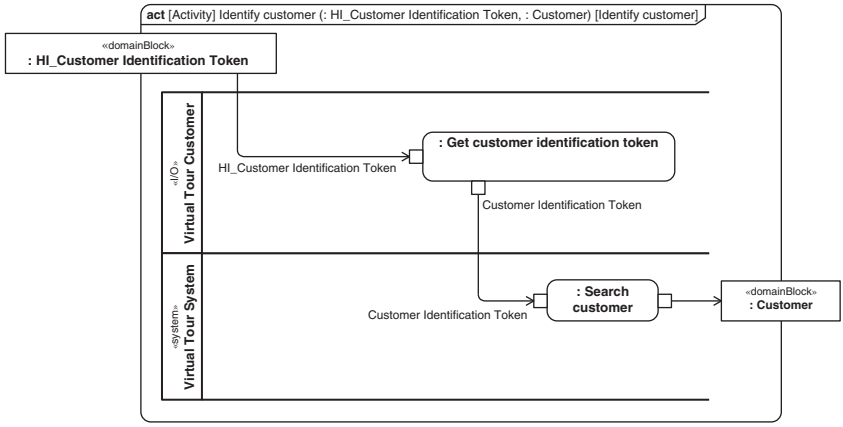


Figure 17.17 Example use case activity "Identify Customer."

In the step of grouping activities, the FAS engine can automatically group all activities that are called from actions in the I/O partitions into functional I/O groups ("Create I/O group" in Figure 17.14). Additionally, a FAS engine could determine potential conceptual mistakes. For example, an activity should not be called from actions in an I/O partition and actions in a non-I/O partition.

Furthermore, the FAS engine can group all nodes of the activity tree, except the leaves, into one "Top Level Functional Group" (second action in Figure 17.14). This ensures that top-level activities in the tree are assigned to a group and that the functional block for representing the whole system of interest itself will be created later and traced to that group.

The two mentioned groups, "I/O" and "Top Level Functional Group," have been created, and activities from our example system have been automatically assigned

to the functional groups by means of trace relationships, pointing toward these activities.

17.6.2 Changing and Adding Functional Groups

The architect will now manually edit the functional grouping (last step according to Figure 17.14). Additional functional groups will be created manually according to the heuristics mentioned further above in Section 17.4, and the architect will have to assign the activities to them, until the assignment of activities to groups represents a disjoint decomposition of the set of activities. A typical result of such work is shown in Figure 17.3 on page 240.

17.6.3 Creating Functional Blocks and their Interfaces

Functional blocks are created automatically by the FAS engine (the first step in Figure 17.15): For each functional group, one functional block of the same name is created. The functional block is linked to the functional group it originated from by a trace relationship. For each block, the FAS engine creates parts by a composition relationship from the functional block that represents the whole functional architecture of the system the block. Furthermore, if the system architect created a composition hierarchy of functional groups, it can also be created automatically between the functional blocks.

Interfaces (interface blocks, ports, and their connections in Figure 17.11) can be created with tool support (second step in Figure 17.15), because interfaces between functional blocks depend on the object flows in the underlying use case activities. The FAS engine can verify by itself if there are object flows between activities behind different functional blocks, and if there are, it can suggest creating an interface between the corresponding blocks.

The architect may now modify the created functional architecture (last step in Figure 17.15). The result of such work is shown in Figure 17.11: While most blocks and ports are unchanged compared to a first version of the functional architecture that had been synthesized with the FAS engine, some additional connections and ports toward system actors have been modeled. In this step, the architect can also create sub-elements of functional elements, now considering not only functional requirements but also nonfunctional requirements.

17.7 Mapping of a Functional Architecture to a Physical Architecture

A functional architecture itself cannot be implemented. Therefore, a physical solution providing the identified functions is needed before the architecture can be

realized in a system. This solution comes with a structured view, which we call physical architecture description. We differentiate two special physical architectures: the logical architecture and the product architecture.

A fully-fledged system model has a base, a functional, a logical, and a product architecture. The functional architecture is allocated to the logical architecture, and the product architecture is derived from the logical architecture (see Chapter 5.4).

In practice, there is often only one physical architecture that has aspects of the logical and the product architecture, which means some blocks represent technical concepts while other blocks represent concrete technical assemblies. A strict separation of the two architecture kinds is useful for example, to get a logical architecture for reuse in other system development projects.

Possible procedures for realizing functions in a physical system have been described in the literature ([103]: 397; [29]; [140]; [253]; [199]: 203). System architects are interested in the allocation of elements in physical architecture from functional elements. See Section 11.9.1 for more information about the functional-to-physical mapping.

In SysML, the physical architectures are modeled with block definition and internal block diagrams similar to the modeling of the functional architecture whereby the blocks now represent physical entities of the system such as software, electrical, or mechanical parts. SysML provides a allocate relationship to model allocations from functional to physical blocks. That relationship can be best modeled and viewed with a matrix representation (Figure 17.18).

For example, the functional part property “customer management” from Figure 17.11 on page 250 can be allocated to a physical part property “virtual tour server application” of a physical architecture of the Virtual Tour system – meaning that the “customer management” functionality is realized with an application that runs on the server (Figure 17.18). If the mapping is valid in any context, you can also allocate the type of the functional part to the type of the physical part, which means the functional block to the physical block. The allocation between the properties is only valid in the contexts that define the properties.

One functional architecture can map to different physical architectures. The functional architecture still remains the same. For example, in tradeoff studies to assess several physical architectures for the same functional architecture or for product families that share the same functional architecture.

The mapping of functions to physical components requires the experience of the architect. In most cases, system projects do not invent completely new technologies but improve existing systems with partial new components. Therefore, the mapping of most functions to physical components is well known. However, the holistic deployment is rarely documented and prevents optimization strategies. We have observed that system architects tend to have the right gut feeling

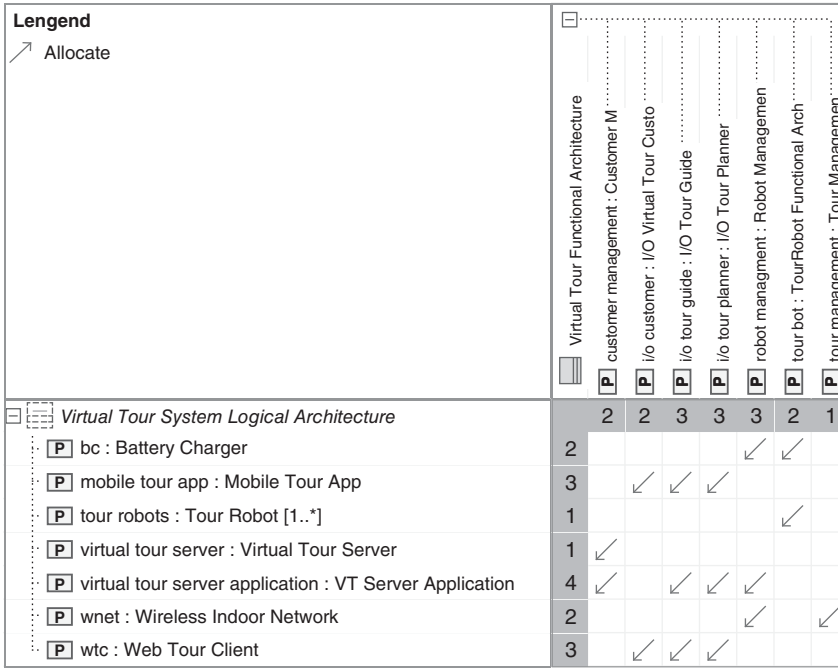

Lengend
Allocate
Virtual Tour Functional Architecture
customer management : Customer M
i/o customer : I/O Virtual Tour Custo
i/o tour guide : I/O Tour Guide
i/o tour planner : I/O Tour Planner
robot managment : Robot Managemen
tour bot : TourRobot Functional Arch
tour management : Tour Managemen
Virtual Tour System Logical Architecture
2 2 3 3 3 2 1
bc : Battery Charger 2
mobile tour app : Mobile Tour App 3
tour robots : Tour Robot [1..*] 1
virtual tour server : Virtual Tour Server 1
virtual tour server application : VT Server Application 4
wnet : Wireless Indoor Network 2
wtc : Web Tour Client 3

Figure 17.18 Functional to physical mapping matrix.

with regards to finding appropriate functional–physical mappings. The quality requirements (for example, performance, efficiency, reliability, maintainability, usability, security), and constraints (for example, project budget, time-to-market, regulatory conditions, available resources) also have a major influence on the design decisions for the physical architecture.

Typically, it is desirable that a functional element is allocated to only one physical element, and a physical element implements only one functional element. In a matrix like the one in Figure 17.18, you can easily identify "hot spots" of one to many allocations. If a functional element is allocated to many physical elements, a coherent functionality is spread over the system. That is not bad per se but requires the attention of the system architects. The same if a physical element implements many functional elements.

17.8 Experiences with the FAS Method

After its first publication in [153], the FAS method has been used in multiple industry projects (for example, [61, 62, 154]), and in the work of researchers and the

systems engineering community (for example, [83, 144, 279]). For the practical application of the FAS method in an industrial project, the procedure according to Figure 17.13 has to be mapped to the processes of the project and the organization. The different analysis and architecting tasks involved in the FAS method are allocated to existing roles. Typically, the use case analysis will be the task of a system analyst or requirements engineer, whereas the creation of the functional architecture will be a system architect's task.

Use cases are modeled in close collaboration between the requirements engineer and the stakeholders of the system. During that phase, it has been observed by the authors that stakeholders can be overwhelmed by the rather technical representation of detailed activity diagrams. Therefore, we prefer to make a nonformal version of these diagrams that focus on the requirements before doing a more formal refinement.

The step of separating I/O actions from core actions is a step that we consider as a good candidate for being carried out in close collaboration between requirements engineers and architects. One of the authors observed that the transformation between activity diagrams from use case analysis and block diagrams of the functional architecture facilitates the communication between requirements engineers and architects: While analysts seem to have their strengths in explaining the expected system functionality with activity diagrams, architects seem to perform best when assessing block diagrams. The FAS method enables both requirements engineers and architects to work in "their" respective world and synchronize each others' findings via architecture generation by means of the FAS method.

In one case that was observed during an industrial project, the architect immediately spotted a concept error after having synthesized the functional architecture with a FAS engine. This concept error had not been found before though the same architect had reviewed the corresponding activity diagrams. The error found in the functional architecture could be easily traced back to the activity model from which it originated by means of the trace relationships that had been created by the functional grouping and by the architecture creation. The error was then corrected in the activity diagram, and the functional architecture was again synthesized. Such fast iterations of architecture synthesis and resynthesis may only be efficient if a FAS engine is available for the used modeling tool.

The paper [62] discusses experiences with the FAS method in different real-world projects. All projects concluded that the FAS method was worthwhile. Most of the systems of the projects have many nonhuman actors. Although the FAS method is based on use case analysis that is more known for interactions with human users, it is also applicable for networked systems as well. Further observations reported in the paper state that the FAS method costs not much effort but leads to a valuable benefit and that the FAS method could also be well applied by less experienced people.

17.9 FAS Workshops

In a FAS workshop requirements, engineers and system architects together create a first version of a functional architecture or a larger update of an existing one. A perfect setting of a FAS workshop is four to eight participants, an even mix of requirements engineers and system architects, and a timeframe of one day. We do not recommend using a modeling tool in a workshop environment with more than three people. It is hardly possible to incorporate everyone in an efficient process in such a scenario. Instead, we propose a card technique to elaborate the functional architecture in an efficient way with a group of people.

The FAS workshop technique can be used in on-site and in online settings. For an online setting, you need in addition to the standard web conference tool an online board collaboration tool that allows the editing and free placing of virtual cards on it.

The procedure of the workshop includes the following steps. Some steps may be skipped if the information is already there. For example, if the workshop should not create an initial functional architecture but a large update of an existing architecture. In that case, you should consider a brief presentation of the existing information for the workshop participants instead.

- Give a presentation for a brief overview of the FAS method and the action items of this workshop.
- Identify the system context and draw the context diagram on a flip chart. The chart should be visible in the workshop location at any time.
- Identify use cases of the system based on the requirements and the system context.
- Write each identified use case with a unique ID number on a single card and pin the card on a wall (Figure 17.19).
- Describe the use case activities and their input and output data. Write each activity on a card with the input data on the left side and the output data on the right side, or, alternatively, on the top and bottom side. Each activity gets a unique sub-id according to the use case, for example, "#8.5" for the fifth activity in use case #8 (see Figure 17.20). Pin the cards on the wall below the corresponding use case cards. Mind that the order of activity cards does not necessarily need to be the order of execution since it is not important for the functional architecture.

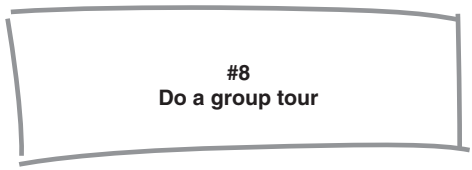


Figure 17.19 Example of use case cards.

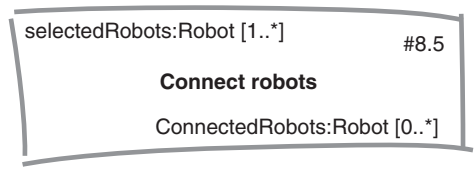


Figure 17.20 Example of a activity card.

- Find the functional groups with the help of the heuristics from Section 17.4. Write each functional group on a card and pin it on the wall. Again write a unique ID for the functional group on each of the cards. Put the grouped activities below the functional group card. The activity IDs on the cards ensure traceability to the associated use case.
- Finally, you can sketch a functional architecture as an internal block diagram on a whiteboard or flip chart. Each functional group is mapped to one part in the functional architecture. It is not necessary to explicitly define the functional blocks in a block definition diagram during the workshop. They are implicitly defined by their usage in the internal block diagram. The type of the parts, that is, the functional block, should have the same name as the corresponding functional group. In addition, you could reference the unique ID of the functional group to be able to change the name without losing traceability. Connect the parts by analyzing the inputs and outputs of the included functions. Use ports at the parts for the connection points and give them proper names. They are the functional interfaces. If it is important, you can further describe the flow properties that is specified by the functional interface on a separate flip chart. Typically, this task can be done outside of the workshop by the system architects in the modeling tool.
- Discuss the architecture. Is it conform to your architecture principles? Seems it proper according to your functional and nonfunctional requirements?
- Take photographs of the workshop results and assign an action to a system modeler to incorporate the results into the system model.
- Ask the workshop participants for feedback on the updated system model with the functional architecture.

17.10 Quality Requirements and the Functional Architecture

The functional architecture is derived from functional requirements via use cases. In this way, most functional requirements are well covered by the functional architecture.

But what about the quality requirements? Some of those requirements are directly related to functional requirements. They call for the quality of the

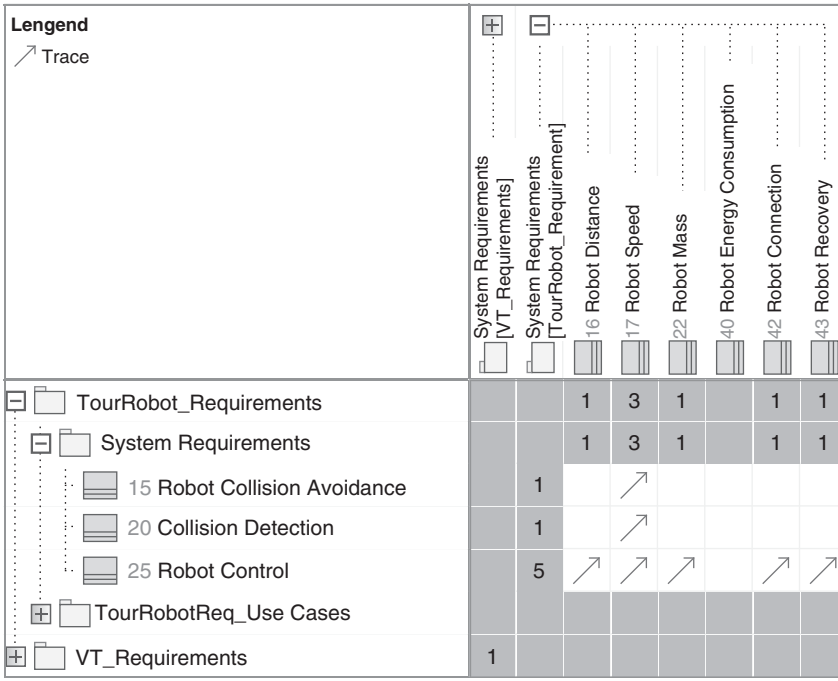


Figure 17.21 Relationship of functional and quality requirements.

functions, for example, the duration or resources needed by a function. The requirements engineer elaborates a matrix that represents the relationships between functional and quality requirements (Figure 17.21).

Since the functional requirements could be traced to the functional architecture, there is also a traceability from the functional architecture to the quality requirements that have relationships to functional requirements (Figure 17.22).

Some of those nonfunctional requirements could be directly incorporated in the functional architecture. Figure 17.23 shows an example of a constraint to satisfy a nonfunctional requirement in a functional architecture. The nonfunctional requirement states that the system must not produce illuminances equal or greater than 250 lux to protect light-sensitive museum artifacts. The light is an object flow in the appropriate use case activities, and the functional architecture has ports with the light as a flow property. An additional constraint at the port ("self.light < 250") specifies that the illuminance of the light at the artifact is always < 250. The constraint is linked with the nonfunctional requirement by a satisfy relationship.

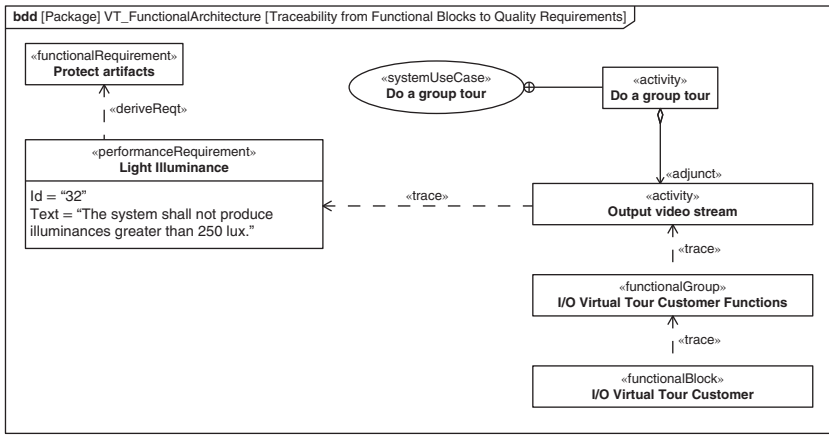


Figure 17.22 Traceability from functional architecture to nonfunctional requirements.

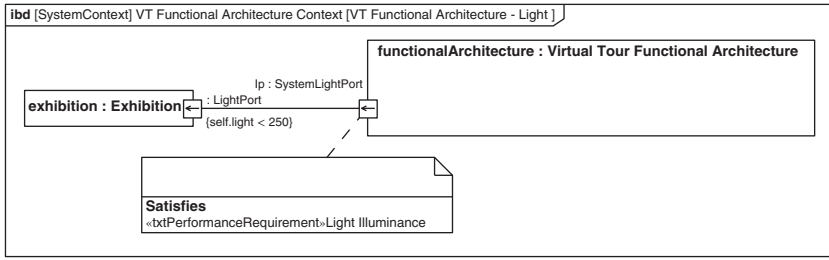

ibd [SystemContext] VT Functional Architecture Context [VT Functional Architecture - Light]
functionalArchitecture : Virtual Tour Functional Architecture
lp : SystemLightPort
exhibition : Exhibition
: LightPort
{self.light < 250}
Satisfies
«txtPerformanceRequirement»Light Illuminance

Figure 17.23 Nonfunctional requirements in the functional architecture.

The "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification" defines a set of UML extensions to represent the quality of service and fault-tolerance concepts [254].

There are also nonfunctional requirements that do not have a direct relationship to a functional requirement, and therefore they are not covered by the functional architecture. For example, the nonfunctional requirement that the color of the housing of the tour robots must be configurable or that the mass of a single robot must not exceed 35 kg. Theoretically, you can treat color as a function that transforms light or as a function for beauty. However, usually, that is not valuable in practice. In consequence, there are some nonfunctional requirements that are not covered by the functional architecture. Be aware of them and consider them in the logical or product architecture.

17.11 Functional Architectures and the Zigzag Pattern

Functional architectures are on different levels according to the SYSMOD zigzag pattern (see Section 9.1). If you decompose a function of a functional block, you need technical decisions at a certain level. Otherwise, further decomposition is not possible. The technical decisions could lead to new requirements and functions that directly depend on the technical decisions. The zigzag pattern describes different levels of requirements and architectures.

Figure 17.24 shows the collision avoidance requirement of the tour robot. The requirement is refined by some use cases, for example, "Steer the robot," as shown in Figure 17.24. Some of the activities of that use case are grouped in a functional group "Robot Control," which leads to a functional block of the same name. The functional block is allocated beside others to the "Anti-Collision System" (ACS). The ACS is modeled as a variation point to specify different variants of the ACS.

The variant of a camera-based collision system leads to new requirements and opens the next level of the zigzag pattern (Figure 17.25). The camera-based ACS is a specialization of the variation point ACS. It has its own functional architecture with functional blocks "Imaging" and "Communication." The functional blocks have relationships to the functional blocks on the upper zigzag level.

You can treat the ACS like a system with its own requirements, use cases, and architectures, including a functional architecture.

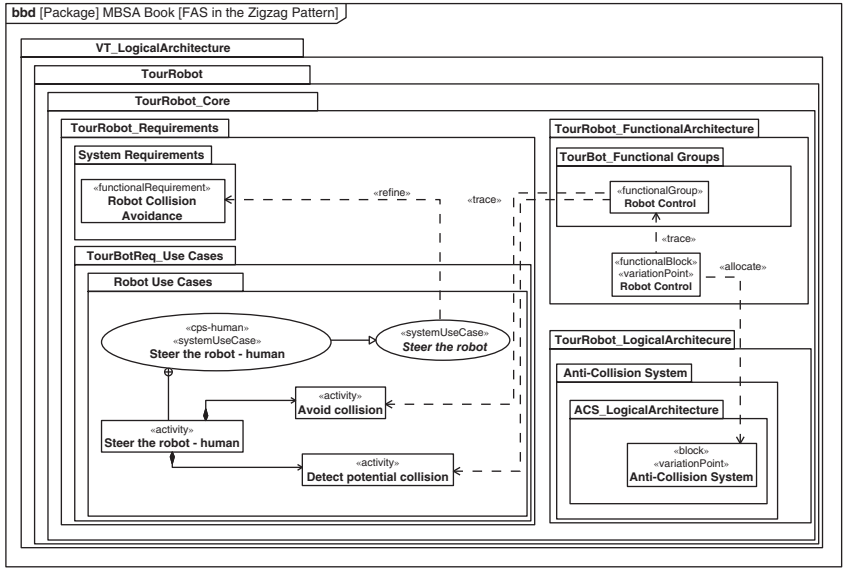


Figure 17.24 Example FAS in the zigzag pattern – level <n>.

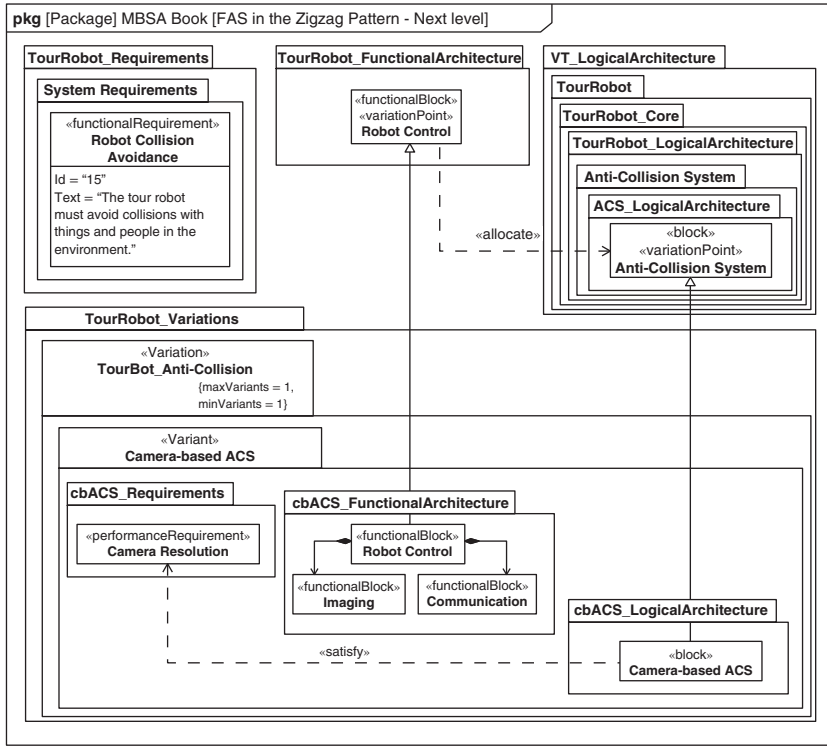


Figure 17.25 Example FAS in the zigzag pattern – level <n+1>.

17.12 CPS-FAS for Cyber-physical Systems

CPS-FAS for Cyber-physical systems (CPSs) is an extension of the FAS method presented the first time in [64].[2] CPSs connect the physical world with the virtual cyberspace and are able to communicate with other systems [126]. The connection of multiple CPSs enables use cases that would not be possible with one CPS alone. The resulting set of interconnected CPSs is called the aggregated system. The functional architecture of the aggregated system is connected with the functional architectures of the CPSs. The method extension CPS-FAS describes how to model the connection of the functional architectures, including the preparation of the CPS models.

2 By the time of writing this book, a more extensive publication of the method is under preparation. We recommend the reader to look for newer publications by the time of reading the book.

The Virtual Tour system we have used as an example so far is a CPS. The manufacturer evolved it from a virtual museum tour system to a more general Virtual Tour system that also includes, for example, company tours. Finally, the manufacturer added machine access to the system because a company specializing in rescue missions wants to integrate the tour robots into their system. The rescue system is an aggregated system. Besides others, it also integrates a surveillance camera system from another company. See Chapter 2 for the whole story of the virtual tour and rescue system.

CPS-FAS proposes two stereotypes, "cps-human" and "cps-machine," for use cases to separately specify the usage of system functions by a human and by a machine actor. Figure 17.26 shows those use cases of the virtual tour robot. Originally, the system had only the use case "Steer the robot" that specified the steering of a tour robot by a human. With the extension to a CPS, this functionality is also made available to machines. The use case "Steer the robot" keeps the common functionality, and the specific human and machine use cases are modeled as specializations of it. The CPS-FAS stereotypes mark the appropriate use cases.

The same principle is applied to other use cases that can be performed by humans and machines. The system also gets some additional use cases only for the machine access, for example, for the management of the machine access like "Set up robot cloud," "Register tour robot," or "Manage cloud access."

The activities behind the machine and human use cases are considered in the functional architecture using the same approach as described above.

The rescue system is a machine actor of the virtual tour CPS. It is also a machine actor of the surveillance camera CPS. Figure 17.27 shows the aggregation of the systems to an aggregated system "Scalable Observation and Rescue System." The CPS integration is modeled by a shared aggregation instead of composition because they are not owned but only used parts of the system.

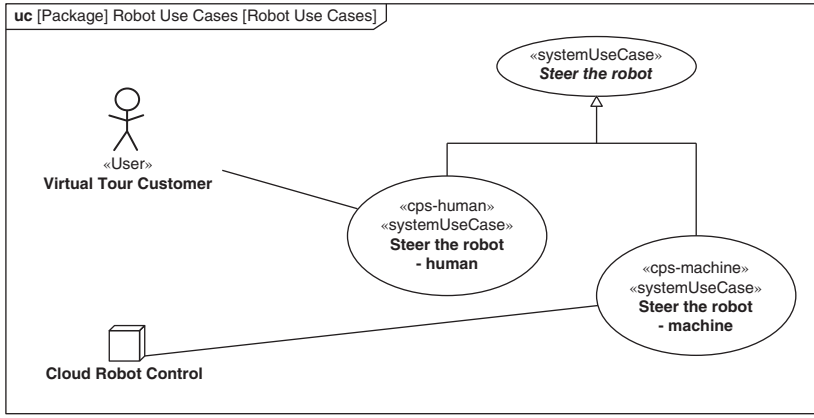


Figure 17.26 CPS use cases for human and machine.

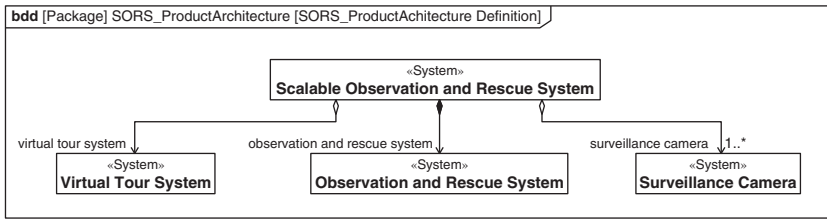


Figure 17.27 Aggregated system.

The integration requires a direct access of the aggregated system model to the virtual tour and the surveillance camera models. Nowadays, this is very unusual, in particular, because there are three different companies involved. In the future, with better technical and organizational interoperability capabilities, it may be a more realistic scenario.

If the aggregated system does not have access to the CPS models, you create proxies that reference the real elements. That means you create a model element in the aggregated system model and reference to the appropriate element in the CPS model, for example, simply in the documentation field of the model element, or you create the link by an adapter technology. In Figure 17.27, the system blocks "Virtual Tour System" and "Surveillance Camera" could be such proxy elements.

Figure 17.28 shows the use case "Observe rescue area" of the aggregated rescue system. It includes the used use cases of the CPS. The include relationship is

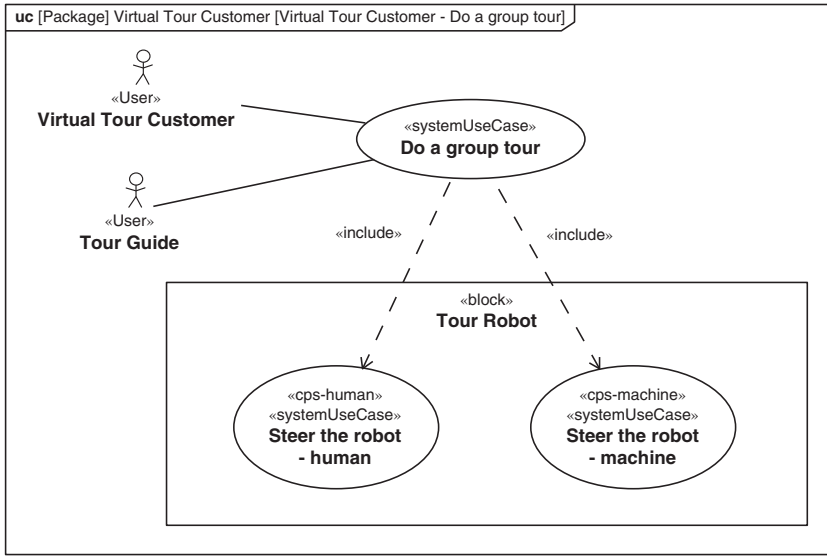


Figure 17.28 Aggregated system use case.

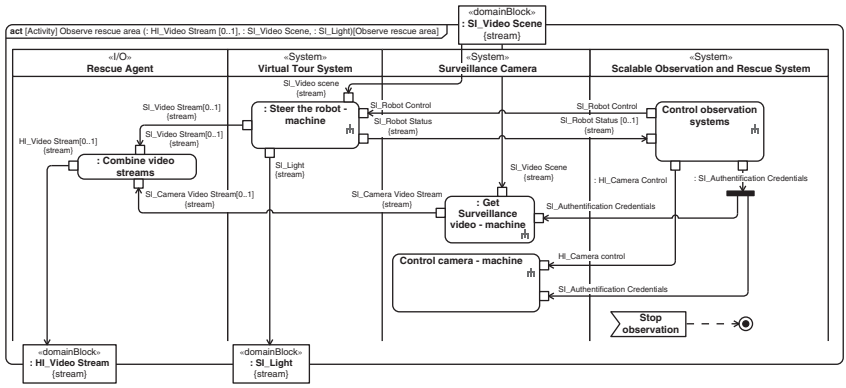


Figure 17.29 Aggregated system use case activity.

reflected in the use case activity in corresponding call behavior actions, as also shown further below. The use case specification also covers the subject of the use case, which is depicted in Figure 17.28 by the box around the use cases. To complete the puzzle, the actors are also linked. For simplicity, we take the whole aggregated system that specializes the particular machine actors of the CPS. Alternatively, it could be the subcomponent of the aggregated system that is responsible for communication to the CPS.

In the use case activity "Observe rescue area" (Figure 17.29), the action in the partition "Virtual Tour System" calls the use case activity of the use case "Steer the robot – machine," and appropriately the actions in the partition "Surveillance Camera."

The use case activities of the aggregated system are transformed into a functional architecture using the FAS method. The activity partitions representing

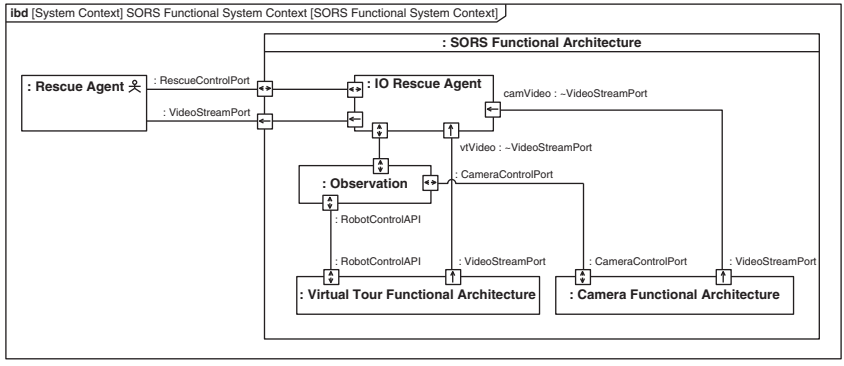


Figure 17.30 Aggregated system functional architecture.

a CPS result in their own functional group or functional block, similar to the I/O partitions. This is the functional architecture of the CPS, which is thus embedded in the functional architecture of the aggregated system (Figure 17.30). At this point, it should be remembered once again that in practice it is probably not possible to access the corresponding model elements directly and that the reference must be established with proxy elements.

18

Product Lines and Variants

Many systems exist in different configurations: a product line, a customized product, or different designs for trade studies. Typically, a single variant of a system affects only a few parts of the system. It is a slight derivation from the core system. However, it is impossible to quantify the number or level of detail that could vary to be still a variant of a system and not a completely new system.

This chapter provides a brief introduction to a broad field along with a couple of ideas. The broad field of product line engineering (PLE), model-based product line engineering (MBPLE), and variability management continue to evolve, and new literature is being published continuously [10, 87, 121].

A car and an aircraft could be a variant of a transportation system. In most cases, it makes no sense in practice to handle a car and an aircraft as variants of the same system and to manage all the appropriate relationships in a system model. The common parts of a transportation system are too abstract. Unfortunately, you cannot measure abstraction, and we cannot give an objective metric. You must decide if the abstraction levels of the common parts and the abstraction level of the variant parts are close enough to be valuable to be part of the same model. The benefit must be larger than the effort to manage the model.

The description of variants is a sophisticated task. It is already challenging to create a valuable model of a single system. Each variation adds another dimension to a multi-dimensional system model. For example, an engine can be a variation point of a car system with three possible variants: diesel, electric, or hybrid engine. The next variation could be the chassis: small, deluxe, cabrio. Now, you can combine the variants, for example, a car with diesel engine and a small chassis, or a car with a hybrid engine and a deluxe chassis, and so on. Any additional variation increases the dimension and the number of potential combinations, and the number of variants rises to a number beyond millions.

In the following, we will first give some definitions and then describe a concept for variant modeling with SysML. Parts of this chapter are based on the book

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

"Variant Modeling with SysML" by our author Tim Weilkiens [270]. We omit the citation of the book at each statement that is taken unchanged or updated from that book.

18.1 Definitions Variant Modeling

We give some brief definitions of the terms how we use them in the context of variant modeling. Our concept of variant modeling is described in the SYSMOD methodology [267, 271] and later refined in the book "Variant Modeling with SysML" [270]. The terms and concepts are conform with common variant concepts presented in publications about variant modeling, for example, the Orthogonal Variability Model (OVM) [202], or ISO 26550 [107] and ISO 26580 [108].

Figure 18.1 shows the domain model of the variant modeling approach. We differentiate between core and variant elements. A core element is used in all system combinations and is independent of any variant elements. A variant element only occurs in some configurations and is part of a variant.

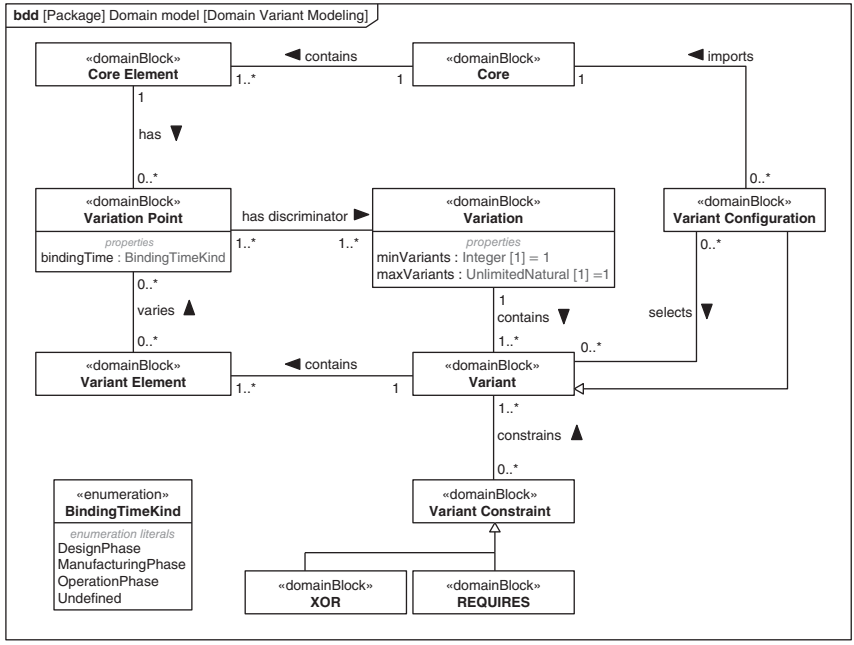


Figure 18.1 Domain model for variant modeling.

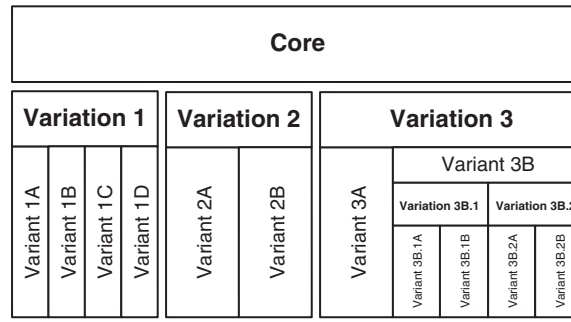


Figure 18.2 Core and variants.

A variation point marks a core element of the system as a docking point for a variant element for example, the engine of a car. It is a core element of a logical architecture since any of our cars owns the technical concept of an engine. We vary that part of the system to provide different kinds of engines. Therefore, the engine is a variation point in the core.

The reason for a variant is called variation. A variation contains a set of variants that have a common discriminator. In our car example, the engine type is a variation. It is the discriminator that distinguishes the variants.

A variant is a complete set of variant elements that varies the system according to the variation. A variant realizes a feature of the system. The diesel, electric, and hybrid engines are variants of the variation engine kind.

A variant configuration is a valid set of variants combined with the core. For example, a car with a hybrid engine and a deluxe chassis. A variant configuration is also a special variant and part of a variation. In our example, the variant configuration "hybrid engine with deluxe chassis" is part of the variation "eco editions."

A variant constraint specifies rules for a valid set of variants. Two common variant constraints are predefined in Figure 18.1. XOR is used to exclude a variant if a specific variant is selected. REQUIRES specifies that other variants are required if a specific variant is selected; other possible constraints are XOR IF or REQUIRES IF.

A variant could again include variations (Figure 18.2). The structure of these variations is the same as for the top-level variations. This recursive structure makes the variant modeling concept scalable for any size of a system.

18.2 Variant Modeling with SysML

SysML v1 does not provide explicit built-in language constructs to model variants. It will be part of the new version SysML v2. However, SysML v1 is useful to model variants, and you can use the profile mechanism of SysML to extend the language with a concept for variant modeling as described above. SYSMOD

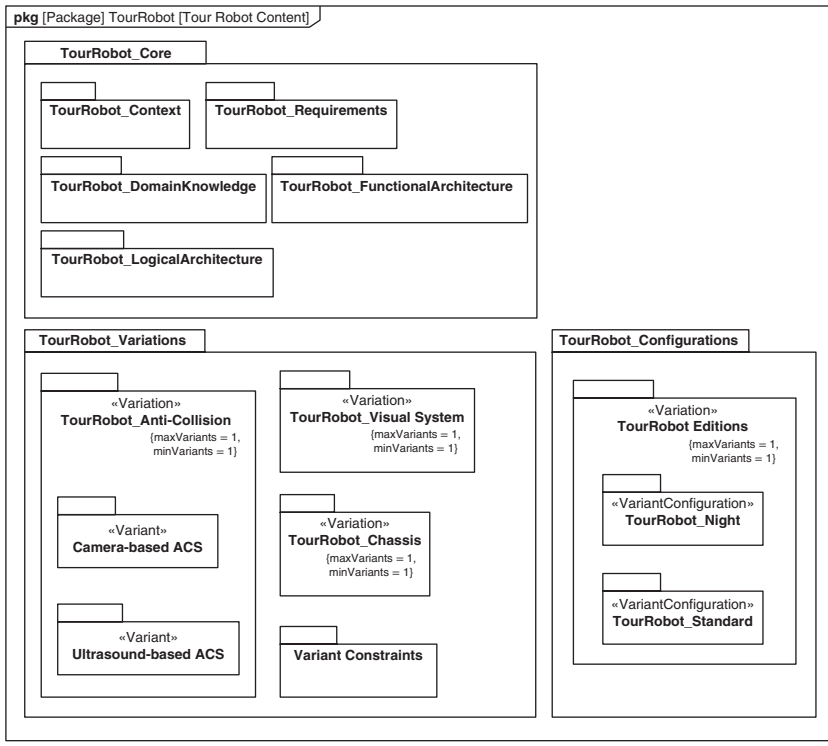


Figure 18.3 Top level package structure.

defines a profile for SysML that includes variant modeling concepts as presented in Section 18.1 [267]. The profile provides stereotypes to model variants, variations, variation points, variant elements, variant configurations, and variant constraints.

Our virtual tour example system has different variations, for example, the anti-collision and the visual system of the tour robots. Figure 18.3 shows the top-level package structure of our model with the packages for the variant modeling. On the first level, we have three packages:

- The configurations package ("TourRobot_Configurations") contains the variant configurations, that means valid sets of core and variant elements combined to a system or system assembly. Since a variant configuration is also a special variant, we have variation packages on the first level and the variant configurations as variants on the next level. The structure of variation and variant packages is described further below.
- The core package ("TourRobot_Core") contains all core elements. The structure of the sub-packages conforms to the system model structure presented in Section 9.9.

- The variations package (“TourRobot_Variations”) contains all variations with their variants. In Figure 18.3, you see three variations: “TourRobot_Anti-Collision,” “TourRobot_Chassis,” and “TourRobot_Visual System.”

A variation is a package with stereotype «variation». Each variation package contains the variants according to the variation discriminator.

The variation package has two additional properties (“minVariants” and “maxVariants”), constraining the number of variants of the variation that can be selected for a single variant configuration. Figure 18.4 depicts the package structure in a SysML package diagram with the “minVariant” and “maxVariant” specifications shown in the package symbol of the variations. For the anti-collision system (ACS), we allow only one variant per variant configuration (“maxVariants=1”). If, for example, “maxVariants” would be 2 it is allowed to implement two different ACS into a single tour robot. The ACS is mandatory. That is specified by “minVariants=1,” which means at least 1 variant must be selected for a valid variant configuration.

A variant is a package with the stereotype «variant». The variant package is the root for all variant elements. They are organized like the recursive package

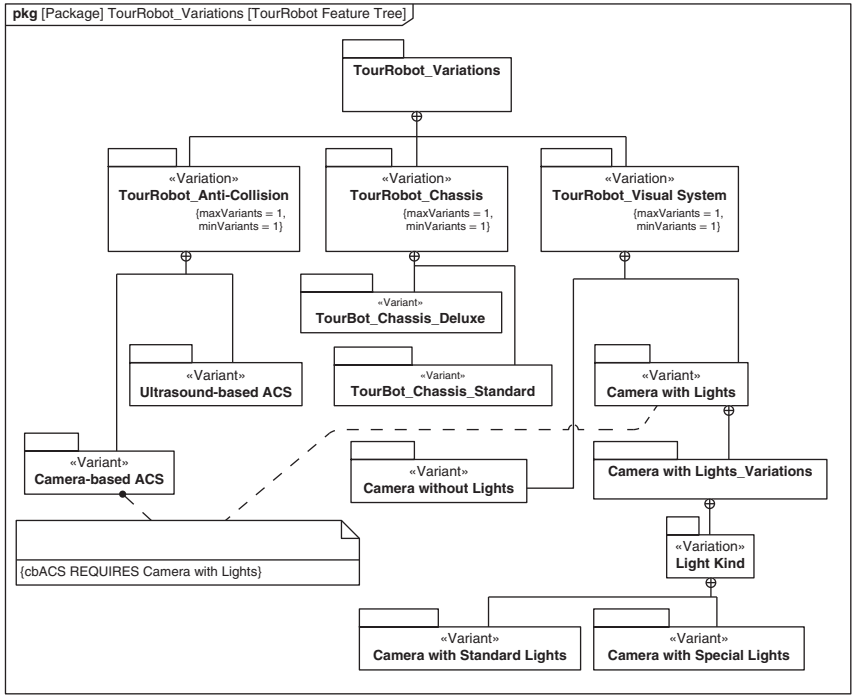


Figure 18.4 SysML feature tree with variants.

structure for system models (Section 9.9). A variant could be handled like a system or subsystem with a context, requirements, architectures as well as again configurartions, variations, and variants.

Maybe you already know the feature trees to describe variations, for example, from the Feature Oriented Domain Analysis (FODA) [135]. You can find more details about FODA and an example of a feature tree in Section 18.3. You can also use SysML with the variant profile to create feature trees, as depicted in Figure 18.4. The notation is different from FODA, but the semantic is conform to FODA trees. Feature trees specify rules between variants like that selected variants must be mutually exclusive.

XOR and REQUIRES are special variant constraints to model rules between variants of the same or different variations. Besides the ACS, the tour robot has variants for the visual system and the chassis. A camera system without additional lights, a camera system with standard Lights, and a camera system with special lights to protect sensitive exhibition artifacts, for example, in a museum. A tour robot can only have one of these visual systems. The camera-based ACS requires a camera with lights to assure the detection of obstacles even in dark areas of the building. Figure 18.4 depicts the appropriate variant constraint. The REQUIRES constraint assures that a camera-based ACS is always combined with a visual system with lights. Since we have two visual system variants with lights, we have introduced another variation "Light Kind." Alternatively, we could introduce a new variant constraint (A REQUIRES B) OR (A REQUIRES C). The additional variant level is more flexible because it is not necessary to change the variant constraint if we define another visual system variant with lights and must combine A, B, C, and D in a single constraint. Formally, the variant constraints are stereotypes («xor», «requires») of the general SysML constraint element.

The chassis variants in Figure 18.4 depend on business requirements. The deluxe chassis is more durable and necessary for intensive use of the system. The standard chassis is for normal use. The relationship between the variants and the business requirements is in the model but not depicted in Figure 18.4.

There are relationships between variant and core elements. The variant elements always depend on core elements and not vice versa. Typically, the relationship is a generalization, which means a core element is a generalization of a variant element. Respectively, the variant element is a specialization of the core element.

Figure 18.5 depicts the variant element "Camera-based ACS" as a specialization of the core element "Anti-Collision System," and the variant element "Camera with Light" as a specialization of the core element "Camera." The variant "Camera-based ACS" requires the variant "Camera with Light" which is reflected not only by the REQUIRES constraint but also by the composition relationship in Figure 18.5.

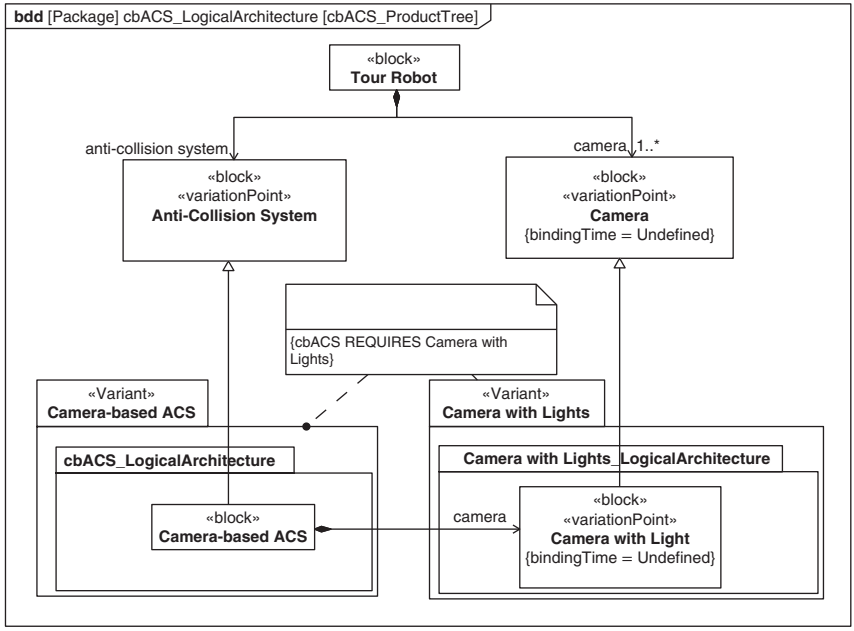


Figure 18.5 Relationship between variant and core elements.

Figure 18.6 depicts the variant configuration "Logical TourBot Night." The main task of a variant configuration is to bundle variants and the core to a valid assembly. The block "Logical TourBot Night" specializes the core element "Tour Robot" from the logical architecture and links the root elements of the variants camera-based ACS and camera with standard lights. Additionally, a variant configuration could define its own structure and behavior. Typically, it is the glue logic to assemble the core and selected variant elements.

The variant configuration is a stereotype «variantConfiguration» specializing the stereotype «variant» and is applied to a SysML package. A variant configuration can be a system, a subsystem, or another system component. Variant configurations are stored as special variant packages below a variation package in the top-level configuration package. Each variant configuration has a package structure like the system model (Section 9.9).

Although all these stereotypes are simple and powerful, it is a challenge to handle the complexity of the model. Even a system model without variants is already a challenge. With variants, it is a multi-dimensional configuration space. Special views, reports, and model transformations are necessary to manage the complexity.

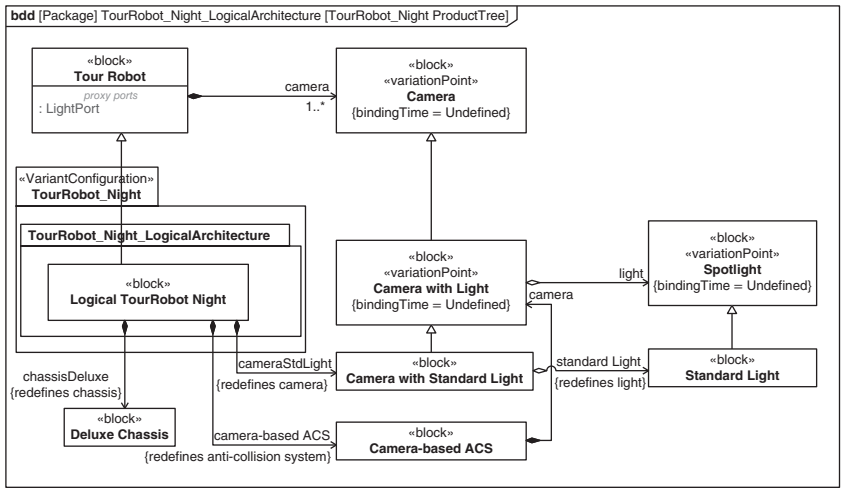


Figure 18.6 Variant configuration.

18.3 Other Variant Modeling Techniques

In this chapter, we will look a little bit out of the box and briefly describe some other variant modeling techniques.

A common technique to model variability is the FODA by Kang et al. [135]. The variability is modeled from the perspective of the stakeholders. It shows the features of the system, their variability, and constraints between the variants. Figure 18.7 shows a FODA feature tree for the Virtual Tour system. It depicts the three features ACS, engine, and human interaction package of the tour robot.

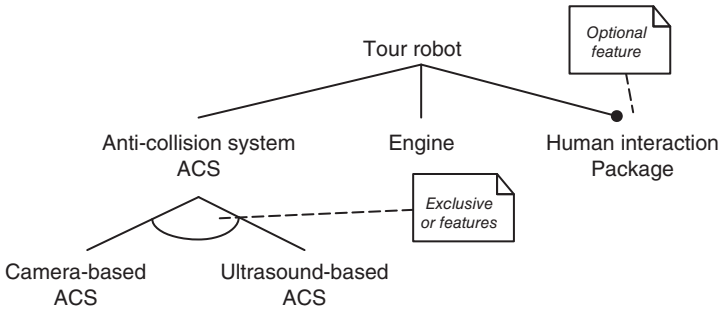
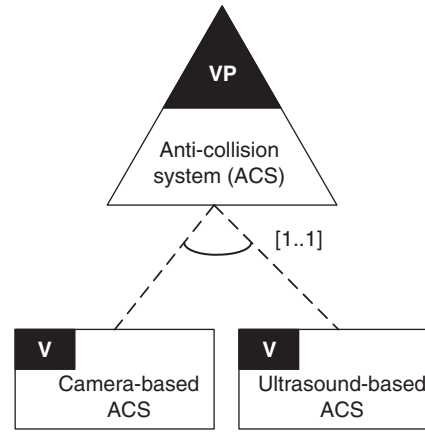


Figure 18.7 Example FODA Tree.

Figure 18.8 Example OVM.



Later Kang extended FODA to the Feature-Oriented Reuse Method (FORM) that, in addition to the requirements and architecture perspective, incorporates the marketing perspective [136].

There are two different basic approaches for variant modeling to integrate the variability information into the system model:

1. Create a separate orthogonal first-class model that specifies the variability.
2. Integrate the variability into the model of the system under development.

Examples for the second approach could be found in [136] and for the first approach in [202]. The approach presented in Sections 18.1 and 18.2 is another example of the first approach. The variability information is separated from the development model and could even be stored in a separate repository.

The OVM [202] provides a modeling language for variability. Figure 18.8 shows an extract of our FODA tree in Figure 18.7 in OVM language.

19

Architecture Frameworks

The term architecture framework is frequently misunderstood. It is often assumed that an architecture framework provides a kind of abstract or common template for a system's architecture. Something like a "skeleton architecture" of a certain kind of system, which only needs to be completed and detailed to get a full-fledged architecture for this system. For example, one could guess that the general architecture for an aircraft, such as body, wings, propulsion subsystem, landing gears, power, and energy subsystem, is abstractly defined in a kind of "Architecture Framework for Aircrafts," and the systems engineer could use that framework as a starting point for an aircraft's system architecture.

But that's not correct.

Of course, there are proven base architectures (see Section C.2), sometimes also called reference architectures, for miscellaneous kinds of systems in the systems engineering domain, but architecture frameworks did not define them!

But what are architecture frameworks instead? The international standard ISO/IEC/IEEE 42010:2011 *Systems and software engineering – Architecture description* [114] defines architecture framework as conventions, principles, and practices to describe an architecture. An architecture framework according to the standard is for a specific domain of application or community of stakeholders.

As a first notice about this definition is that it is about the description of architectures. So that these architecture descriptions comply with certain standards, architecture frameworks provide a set of conventions, principles, and practices which can be used by systems engineers to create such descriptions. Creating and describing a system's architecture from scratch can be a complex and daunting task, and therefore, architecture frameworks can be helpful and could provide a good guidance. They should simplify the process and guide an architect through all areas of architecture development.

Another noteworthy detail is that this definition makes no specific statement about the kind of system whose architecture shall be described. It could be a technical system architecture, a software architecture, an Enterprise

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

Architecture (EA), or a System of Systems (SoS) architecture, just to mention a few possibilities. The definition speaks quite general about "established within a specific domain of application and/or community of stakeholders." What is considered to be "established" in certain domains may be very different.

In practice, architecture frameworks are very well known in the world of Enterprise Architecture development, and System of Systems Engineering (SoSE). (A detailed discussion of the concept of a System of Systems can be found in Section 4.2.) There are a number of different frameworks for SoSE, especially in the defense domain. Before we briefly discuss some of those frameworks, we first have to discuss the term Enterprise Architecture, and afterward, we would like to take you on a short excursion to the typical characteristics of a system of systems.

Before that, a note: Some architecture frameworks come with their own definition of terms. In the following, they will be explained by means of their proprietary terminology, even if some of the terms have a slightly different meaning in the rest of this book.

19.1 Enterprise Architectures

If architecture frameworks are frequently used in the world of Enterprise Architecture development, we first have to define the term Enterprise Architecture (short: EA). In fact, there is no generally accepted definition of the term Enterprise Architecture. Various organizations (public and private) promote their understanding of the term. For example, the *Federation of Enterprise Architecture Professional Organizations* (FEAPO) defines it the following way:

> Enterprise Architecture is a well-defined practice for conducting enterprise analysis, design, planning, and implementation, using a holistic approach at all times, for the successful development and execution of strategy. Enterprise Architecture applies architecture principles and practices to guide organizations through the business, information, process, and technology changes necessary to execute their strategies. These practices utilize the various aspects of an enterprise to identify, motivate, and achieve these changes [74].

But what is meant with an enterprise? Broadly speaking, the term enterprise covers various types of organizations. The size of the organization, its industry, its ownership model (private or public), or its geographical distribution does not matter. Enterprise can mean a small to mid-size company, a multinational corporation, or a huge joint undertaking with a common goal shared by several global players. An enterprise can be a peacemaking and peacekeeping deployment

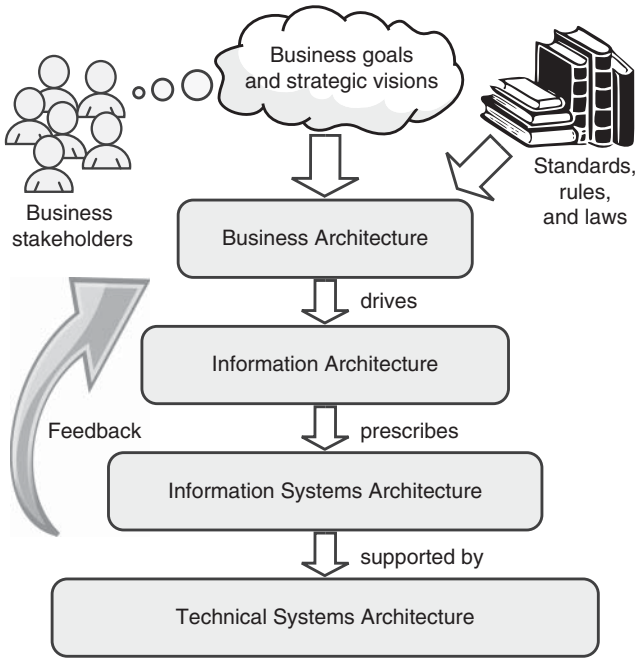


Figure 19.1 The interrelationship between business goals, information architecture, and technology environments.

of North Atlantic Treaty Organization (NATO) forces, or an outer space research program of the NASA. And of course, the management, operation, and further development of a rescue mission from our accompanying case study in this book (see Chapter 2) can also be considered as an enterprise.

Figure 19.1 depicts the way how a technical system architecture is derived from the stakeholder's business goals and visions. It is not sufficient that individual projects or the product development just fulfill immediate stakeholder needs. Instead, a long-term view on all processes, systems, and technologies of an enterprise is required, so that individual projects or the product development can provide general-purpose capabilities. This is important for investment decision-making, work prioritization, and resource allocation. In our world of ever-increasing complexity, ever-changing business environments, and globalization, it is not only of huge importance that an enterprise is able to appropriately respond to disruptive forces. Furthermore, every enterprise has its visions and goals that adapt a predicted future. This requires a holistic view on the enterprise, and a departure from an isolated treatment of individual units, e.g. business segments, single departments, or solely the IT infrastructure.

Enterprise Architecture development is a business-centric approach, and not a technology-centric one.

Enterprise Architectures (EA), and thus the discipline of Enterprise Architecture development, has the following major goals:

1. The architecture of an enterprise, its units, policies, processes, strategies, and technological infrastructure (e.g., IT systems), supports all stakeholders in achieving short and long-term business objectives and visions.
2. EA fosters an alignment of the technological systems developed by resp. used by an enterprise with its business goals and strategic direction.
3. EA helps an enterprise to learn, grow, innovate, and respond to market demands and changing basic conditions.
4. EA fosters and maintains the learning capabilities of enterprises so that they may be sustainable.

Of course, it strongly depends on the domain what is meant by the notions *business objective* and *business vision*. The business objectives of an enterprise in the defense sector, or in the space domain, are usually different than in the commercial business environment (e.g. insurance companies, world of finance, or trading). But on an abstract level, it is all the same: all decisions, activities, developments, and procurements that are done by an enterprise must support these goals, and the discipline of Enterprise Architecture development is the key.

19.2 Characteristics of System of Systems (SoS)

In Section 4.2 in the Chapter on "Systems, Systems of Systems and Cyber-physical Systems," the concept of System of Systems has already been defined and discussed. Therefore, in this section, we would just like to go into a little more detail about the key characteristics that are typical for a System of Systems.

A System of Systems is different from a single monolithic system and from other kinds of system groupings (e.g. a Family of Systems), in the following manner:

1. **Loose coupling of elements:** In general, the term coupling refers to the degree that constituents depend on each other. One distinguishing feature, which lets you know that you are confronted with a System of Systems instead of a single monolithic system, is the remarkable loose coupling of its system elements, i.e. of its constituent systems. The constituent systems usually do not interchange forces, fluids, or large amounts of energy with each other. The coupling between them usually takes place via data exchange through IT interfaces. This is often also accompanied by geographical distribution of the constituent systems.

2. **Emergent behavior**: The System of Systems behavior arises from the cumulative actions and interactions of its constituents. A System of Systems behavior is more than just the sum of the capabilities of its constituent systems. To create emergent behavior, all constituent systems must interact with each other. A prerequisite is that each system can exchange messages with other systems. Since Emergence is a key characteristic of SoS, the term is deepened in the following subsection.
3. **Operational independence:** If a System of Systems is disassembled into its constituents (component systems), these systems are able to operate independently outside of a System of Systems compound, or they can join another System of Systems.
4. **Independent management of elements:** Most of the constituent systems that are building a System of Systems are independently developed, manufactured, purchased, and administrated. Their life cycle is independent from the life cycle of the whole System of Systems and from that of the other constituent systems.
5. **Evolutionary development**: A completely fully developed System of Systems does typically not exist. Its development and existence is evolutionary, i.e. over the entire life cycle constituent systems can be added, removed, modified, or exchanged, depending on new or changed requirements, new capabilities that are required, altered basic conditions, or changed objectives.

Although not every System of Systems have all of the aforementioned properties, these criteria are typical characteristics of them. The most outstanding characteristic of a SoS is its emergent properties. In this respect, a SoS differs most significantly from the concept of a Family of Systems. With a Family of Systems, no capabilities beyond the individual functionalities of the joint component systems are provided by the merger.

19.2.1 Emergence

Emergence was first defined properly by the English philosopher George Henry Lewes (1817–1878) in his 1875 published work *Problems of Life and Mind* [157]:

> Every resultant is either a sum or a difference of the co-operant forces; their sum, when their directions are the same – their difference, when their directions are contrary. Further, every resultant is clearly traceable in its components, because these are homogeneous and commensurable. It is otherwise with emergent, when, instead of adding measurable motion to measurable motion, or things of one kind to other individuals of their kind, there is a co-operation of things of unlike kinds. The emergent is unlike its components insofar as these are incommensurable, and it cannot be reduced to their sum or their difference.

The term Emergence refers to the occurrence of new properties from the interaction of the elements in a system, whereby these emergent properties cannot be derived directly from individual parts of the system. Even with perfect knowledge about the components, some properties of the entire system cannot be predicted. Commonly known is such an effect also from a famous quote attributed to the Greek philosopher and polymath Aristotle (384–322 BCE), "The whole is more than the sum of its parts."

Emergent properties can be very well observed in nature. Good examples are the shape and behavior of a flock of birds or school of fish. From a human's perspective, a flock of birds or a school of fish behaves like a single, large organism. But their individuals only interact with a small amount of other group members in their closest neighborhood. The single bird, or fish, know nothing about the shape and position of the whole swarm. Through emergence collective properties arise, that cannot be localized in the constituents of the swarm.

Another good example for emergent behavior is a colony of ants. The individual ants do not know anything about coordinated foraging. A single ant wanders in search of food around randomly, as biologists have discovered. But suddenly, at a certain point, the movement patterns of the ants change from chaos to order. It seems that the whole colony has a common strategy for coordinated foraging, and the collective behaves as a highly efficient complex network.

Even in the human-made civilization, there are numerous examples of emergence. For instance, the complex business world of today knows and fears emergent properties. There, the term is often associated with unintended effects that may arise from different and superimposed management decisions in large enterprises. Suddenly something happens that no one has foreseen, or more precisely: could not foresee. And these effects are increasingly becoming a problem for traditionally organized, hierarchical organizations, because more and more frequently these effects bring rather damage to the company. The same applies to the globally interconnected, complex financial world.

Even in systems engineering, such arising emergent properties can be both, intended and unintended. Quite often, especially unwanted emergent behavior arises after the integration of a system. All elements that compose the system work correctly for themselves. But if these elements are assembled to a system, this system may show undesirable behavior.

A System of Systems depends on emergent behaviors to achieve its goals. It is explicitly desired that a SoS evolves certain emergent properties. A special challenge for the System of Systems Engineer is to create an architecture in such a way, that these properties will emerge purposefully. That could be a difficult task, because emergent composition is often poorly understood. Just imagine that you should build a highly reliable System of Systems from unreliable constituent systems.

19.3 An Overview of Architecture Frameworks

As already mentioned above, there are a variety of different architecture frameworks in existence. They differ mainly with respect to their field of application. Some have been created in the military environment, some other in the domain of Enterprise Architecture IT development, and some are especially designed for complex SoSE in the space domain.

Several architecture frameworks are defining a set of perspectives and views (The concepts of perspectives and views are discussed in detail in Chapter 11) that should be created by the SoSE. For this purpose, some of these architecture frameworks specify a conceptual, UML-based meta-model which defines appropriate model elements and relationships. The concepts defined in such meta-models are usually available as a stereotype profile for importing into modeling tools (What a stereotype profile is, is described in Section A.6). Some frameworks also specify an enterprise architecture development process, respectively, methodology.

The following subchapters provide an overview and list the most important properties of a couple of best-known architecture frameworks.

19.3.1 Zachman Framework™

Brief description	
First year of publication:	1987
Developed resp. published by:	John A. Zachman, Zachman International, Inc.
Primary field of application:	IT Enterprise Architecture
Development process or methodology:	No
Meta model (Ontology):	No

The *Zachman Framework™ for Enterprise Architecture and Information Systems Architecture* [2], better known under its short name *The Zachman Framework*, was originated by John A. Zachman, an US-American business and IT specialist, in 1987. It is considered as one of the most important frameworks and influenced the contemporary understanding of enterprise architectures significantly. Many subsequent developed Enterprise architecture frameworks have had the Zachman Framework as a foundation.

The framework defines pre-structured views and layers to represent an information technology (IT) enterprise. Unlike similar frameworks that often contain process models, the Zachman Framework does not prescribe any process or methodology. It focuses on the roles involved and assigns them to objects that shall be viewed from different perspectives. The Zachman Framework

thereby provides a comprehensive tool to consider all relevant aspects, from all perspectives, while designing and developing an enterprise IT architecture.

Although this framework plays virtually no role in systems engineering, it is regarded as one of the pioneer works in the area of architecture frameworks.

19.3.2 The TOGAF® Standard

Brief description	
First year of publication:	1995
Developed resp. published by:	The Open Group®
Primary field of application:	Enterprise Architecture
Development process or methodology:	Architecture Development Method (ADM)
Meta model (Ontology):	No

First published in 1995, *The Open Group Architecture Framework* (TOGAF®[1]) was based on the *US Department of Defense Technical Architecture Framework for Information Management* (TAFIM; a framework that is not discussed here). It has been developed by The Open Group, a vendor and technology-neutral industry consortium. The framework provides an approach for designing, planning, implementing, and governing an Enterprise IT architecture. The current version during the creation of this book is The TOGAF Standard 9.2 [245], published on 16 April 2018. Compared to the previous version 9.1 from 2011, probably the most significant change in version 9.2 of TOGAF is the introduction of the so-called TOGAF Library, a structured library of resources that support the TOGAF Standard. Beyond that, however, there were also various improvements and cleanups in the core of the framework.

Unlike many other frameworks, the TOGAF Standard provides a detailed process model. The Architecture Development Method (ADM, see Figure 19.2) is an iterative process over all phases of enterprise architecture development, adaptable to specific needs.

Tailoring the ADM to support specific needs is, among other preparations, something that should take place in the Preliminary phase. In phase A, stakeholders are identified, and the scope, constraints, and expectations are defined. Furthermore, a vision of the project is developed. The core phases of Enterprise Architecture development are the phases B, C, and D. In these three steps, the Business Architecture, the Information Systems Architecture, and the Technology Architecture are developed. Another step (phase F) is planning the migration, i.e. how to move

1 TOGAF is a registered trademark of The Open Group.

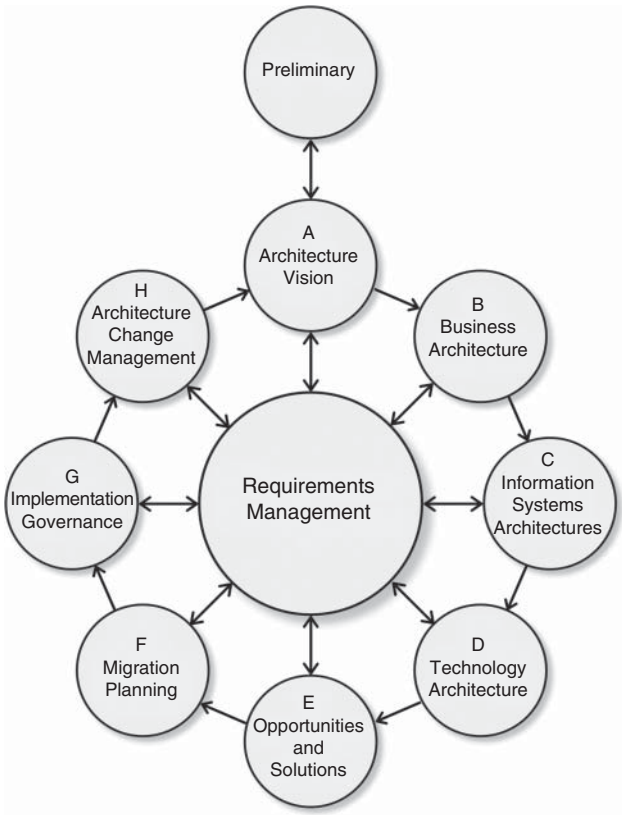


Figure 19.2 The TOGAF® architecture development method (ADM).

from the Status Quo (the current architecture of the enterprise) to the target architecture. Phase G ensures that the implementation project conforms to the planned architecture. Every step has a bidirectional relationship with the Requirements Management in the center of the ADM diagram. This is to emphasize the importance of requirements in each step, and that new or changed requirements can be discovered every time due to new insights.

In contrast, the TOGAF standard does not define or prescribe form and look of its process outcomes, i.e. the deliverables that must be built in every phase during development. For instance, the form how the Technical Architecture in step D has to be documented resp. visualized is not defined. It could be a simple document containing a textual description, a textual and graphical description in a Wiki, or a model of the technical architecture in a modeling tool. For this reason, TOGAF is

often combined with other views-oriented frameworks that have a UML-based meta-model, which in turn does not have a fully fledged process or methodology.

Although the TOGAF standard is preferably intended for the development of enterprise IT architectures, the framework could also be used in the systems engineering domain. For instance, the *Norwegian Armed Forces* chose TOGAF's ADM as their process model, and the NATO Architecture Framework (NAF, see Section 19.3.6) for meta-model and content organization (Views) [132].

19.3.3 Federal Enterprise Architecture Framework (FEAF)

Brief description	
First year of publication:	1999
Developed resp. published by:	U.S. Chief Information Officers (CIO)
Primary field of application:	Enterprise Architecture of Federal Governments
Development process or methodology:	Collaborative Planning Methodology (CPM)
Meta model (Ontology):	Consolidated Reference Model (CRM)

In the United States, the Chief Information Officers (CIO) serve as a central resource for information on Federal information technology (IT). In 1996, the *Clinger-Cohen Act* (CCA), formerly known as the *Information Technology Management Reform Act*, was enacted by the U.S. Congress to reform and improve the way Federal agencies acquire and manage IT resources. Thereupon in September 1999, the CIO published an architecture framework that should support Federal authorities and agencies in the development of Enterprise IT Architectures: the *Federal Enterprise Architecture Framework* (FEAF) version 1.1. In January 2013, a revised version 2.0 of FEAF was published [73].

As its name suggests, FEAF is primarily intended for the Enterprise Architecture development in Federal agencies. The framework should standardize the development and use of architectures within and between those Federal agencies. FEAF provides both, a structure (Consolidated Reference Model) and a methodology (Collaborative Planning Methodology). The Consolidated Reference Model consists of six reference models and provide standardized categorization for strategic, business, and technology models. The Collaborative Planning Methodology is a full planning and implementation life cycle for Federal Enterprise architectures. It consists of two main phases: Organize and Plan, and Implement and Measure. These main phases are further divided into smaller steps. Although the Collaborative Planning Methodology looks strictly sequential and reminds of a waterfall-like approach, it is emphasized that there are frequent iterations within and between the phases.

It is not known whether FEAF was also used for any SoSE architecture development purpose. The development of FEAF demonstrates, however, that very special requirements and needs in a certain domain may result in a highly specialized architecture framework.

19.3.4 Department of Defense Architecture Framework (DoDAF)

Brief description	
First year of publication:	2003
Developed resp. published by:	U.S. Department of Defense
Primary field of application:	Military operations and System of Systems
Development process or methodology:	6-step Architecture Development Process
Meta model (Ontology):	DoDAF Meta Model (DM2), based on UML

The predecessor of the *U.S. Department of Defense Architecture Framework* (DoDAF) was the *C4ISR Architecture Framework* (C4ISR AF, a framework that is not discussed here) version 2.0. C4ISR is a military acronym for command and control, communications, computers, intelligence, surveillance, and reconnaissance. The term refers mainly to the interconnection of all management, information, and monitoring systems in order to create a more accurate picture of the overall situation and thus to improve decision-making and leadership skills.

DoDAF 1.0 replaced its predecessor in August 2003. There followed an evolutionary development of the framework, and ended in DoDAF 2.0 which was released in May 2009. One of the major changes between DoDAF version 1.0 and 2.0 was the transition from a product-centric process to a data-centric process. The current version is DoDAF 2.02 [255].

DoDAF is a view-oriented architecture framework, i.e. it provides an organized, meta-model-based visualization infrastructure for specific stakeholder's concerns. Therefore, the framework consists of eight main viewpoints, whereby the meaning of the term viewpoint is consistent to its meaning in SysML. A viewpoint defines specifies rules for constructing a view (perspective) on the system under development including a set of concerned stakeholders and the purpose to address their concerns.

Each one of the eight viewpoints consists of a coherent set of views representing the architecture of the enterprise from the perspective of the viewpoint. For example, DoDAF's Operational Viewpoint (OV) consists of nine different views (OV-1…OV-6c), all together describing operational scenarios, activities, and requirements that are necessary to develop those capabilities that must emerge through the planned System of Systems. The whole DoDAF 2.0 framework consists of 52 views.

All DoDAF views must follow an UML-based meta-model (DoDAF Meta Model; short: DM2) which defines specific types, semantics, relationships, rules, and constraints about how to build a DoDAF-conformant System of Systems enterprise architecture model. For many well-known UML/SysML modeling tools on the market, this meta-model is available as an UML 2 stereotype profile resp. a plug-in. The *Unified Profile for DoDAF/MODAF* (UPDM, see Section 19.3.9) contains all stereotypes required for DoDAF modeling.

In contrast to its very detailed meta-model, DoDAF contains only a high-level six-step phase model for dealing with the development process of an enterprise architecture. If a more detailed methodology is required, a mapping of DoDAF 2 views to TOGAF's ADM deliverables and activities might be a feasible solution.

19.3.5 Ministry of Defense Architecture Framework (MODAF)

Brief description	
First year of publication:	2005
Developed resp. published by:	British Ministry of Defense (MOD)
Primary field of application:	Military operations and System of Systems
Development process or methodology:	MODAF Architecting Process
Meta model (Ontology):	MODAF Meta Model (M3), based on UML

The *British Ministry of Defence Architecture Framework* (MODAF) [172] is a view-oriented architecture framework that is based on the DoDAF version 1 baseline. MODAF version 1.0 was released in August 2005. The last version 1.2.004 was released in May 2010.

As well as with DoDAF and NAF, each viewpoint of MoDAF offers a different perspective on the System of Systems project to support different stakeholder concerns and interests. MODAF 1.2.004 consists of seven different viewpoints: All Views viewpoint, Strategic View viewpoint, Operational View viewpoint, System View viewpoint, Technical Standards View viewpoint, Acquisition View viewpoint, and Service Oriented View viewpoint. All these viewpoints contain a different number of views, in total there are 47 views.

The MODAF Meta Model (M3) defines an UML2-conformant stereotype profile that specifies the structure of the architectural information that is presented in the MODAF views. Even the standardized UPDM (see Section 19.3.9) supports the modeling of MODAF-conformant enterprise architectures.

MODAF does not prescribe an official architecting process or methodology. However, there is a paper on the MODAF website available that describes a six-step approach named "The MODAF Architecting Process" which is just an example of one way to take.

19.3.6 NATO Architecture Framework (NAF)

Brief description	
First year of publication:	2004
Developed resp. published by:	North Atlantic Treaty Organization (NATO)
Primary field of application:	Military operations and complex System of Systems
Development process or methodology:	Proprietary methodology (NAFv4 Chapter 2)
Meta model (Ontology):	ArchiMate®, UAF

First published in September 2004, the NAF is an Enterprise Architecture framework used by the NATO to define the operational context, the system architecture, and the supporting standards and documents that are necessary to describe an enterprise. The framework is broadly accepted and used in the defense domain. The main objective of NAF was and is the successful exchange of architecture-data between all stakeholders in the context of joint missions of the intergovernmental military alliance NATO. Nowadays, the framework is also used for similar purposes in the civil sector when large joint undertakings are to be carried out.

The first version of NAF was developed on the base of DoDAF (see Section 19.3.4). Like DoDAF, NAF is a views-oriented framework. The current version 4 [47], issued in January 2018, is explicitly designed for both military and business usage. NAFv4 defines a methodology, a set of 47 viewpoints which are organized in a two-dimensional classification scheme called NAF Grid (five rows and nine columns), a meta-model, and a glossary. The framework is designed in such a way that users can extend it according to their own requirements.

NAFv4 is now fully compatible with international architecture standards, such as ISO/IEC/IEEE 42010 [114], 42020, and 42030, with the process model and life cycle stages of ISO/IEC/IEEE 15288 [115], and also with TOGAF (see Section 19.3.2). In order to meet the requirements of use in civilian areas as well, the framework is also compliant with ISO 15704 *Enterprise modeling and architecture – Requirements for enterprise-referencing architectures and methodologies*

[106], a standard that defines concepts, rules and requirements for architecture descriptions for an enterprise in industrial manufacturing, automation, and other civil domains.

Compared to the previous version NAF version 3 with its proprietary meta-model named *NATO Architecture Framework Metamodel* (NMM), NAFv4 decided to adapt the already existing and widely used meta-models of ArchiMate®, which is a modeling language for enterprise architectures and a standard by The Open Group, and the OMG Unified Architecture Framework® (UAF; see Section 19.3.9).

NAF is seen as a key enabler for effective communications about the results of an architecture development process. The ability to effectively federate capabilities in coalition operations is crucial for the success of joint operations of NATO forces. An architecture development using NAF should therefore ensure that various component systems of different armed forces are working optimally together, and let emerge the capabilities of the System of Systems compound as expected by the stakeholders.

As mentioned before is the NAF also suitable for nonmilitary projects and enterprises. One example is the *Single European Sky ATM Research Programme* (SESAR, http://www.sesar.eu). SESAR is a joint undertaking initiated by the *European Commission* and the *European Organisation for the Safety of Air Navigation* (EUROCONTROL). Its goal is to completely overhaul the European airspace and its air traffic management (ATM). SESAR uses a framework during its development phase that is based on NAF and MODAF, but was customized to adapt aspects of them to meet ATM needs.

19.3.7 TRAK

Brief description	
First year of publication:	2010
Developed resp. published by:	London Underground Limited/UK Department for Transport
Primary field of application:	Each kind of complex system or System of Systems
Development process or methodology:	No
Meta model (Ontology):	TRAK Meta Model, based on MODAF Meta Model (M3)

TRAK [200] is an interesting example for an architecture framework that was released under an Open Source License. Its logical definition is released under

the GNU Free Documentation License (GFDL). Implementations of TRAK, e.g. plug-ins or stereotype profiles for modeling tools, are licensed under a GNU Public License (GPL).

Furthermore, TRAK is a framework that is especially targeted to systems engineering. Its development started in 2009 under the working title *The Rail Architecture Framework* because it was thought that the result may be specific to the railway transportation domain. The reason for that was that its development based on the then current views of architectural description within London Underground, a public rapid transit system in the capital city of the United Kingdom which is also known as "The Tube." However, it quickly became apparent that the finally developed framework will be domain-agnostic, i.e. it contains no rail-specific views or meta-model elements. This is one of the reasons why today it is referred to as just "TRAK," and the original working title has become meaningless.

According to their own admission, TRAK conforms to the international standard for architecture description ISO/IEC/IEEE 42010 [114]. Furthermore, TRAK mandates no development process or methodology.

TRAK is intended as a lightweight, pragmatic, and view-oriented framework for each kind of complex system. It has five architecture perspectives (Enterprise Perspective, Concept Perspective, Procurement Perspective, Solution Perspective, and Management Perspective) each of which contains a number of related views. In contrast to the relatively large amount of views in other popular frameworks of the defense domain such as MODAF (47 views) or DoDAF (52 views), defines TRAK only 24 views.

Due to the fact that TRAK is open-source, the whole documentation is hosted on *SourceForge* (https://trak.sourceforge.io), a web-based repository which acts as a centralized location for free and open-source projects, mainly for the software engineering domain.

19.3.8 European Space Agency Architectural Framework (ESA-AF)

Brief description	
First year of publication:	Unknown
Developed resp. published by:	Telespazio VEGA, on behalf of European Space Agency (ESA)
Primary field of application:	Space System of Systems
Development process or methodology:	Based on TOGAF (ADM)
Meta model (Ontology):	ESA-AF meta-model, based on UPDM

The exact release date of this framework is not known. European Space Agency Architectural Framework (ESA-AF) [91] is based on established frameworks, such as MODAF and TOGAF. Its meta-model is based on UPDM (see Section 19.3.9). Due to the fact that MODAF emerged in the defense domain, and TOGAF was primarily developed for Enterprise IT systems and infrastructure in the business domain, both frameworks do not address issues that are crucial for the space domain. Therefore, these known frameworks and methodologies have been tailored and extended to satisfy specific needs that are characteristic for Space SoSE. Some of these special needs are as follows:

- **Regulation needs in a multi-national environment:** In European space programs, usually many European member countries are involved. Each participating nation has its own national regulation regarding governmental policies, processes, and procedures. ESA-AF supports the SoSE process in coordinating and aligning these individual rules and policies so that the space program can be successful.
- **Space domain needs:** The framework supports an accurate representation of space domain concepts and their relationships. For example, these concepts can be represented by space-specific types for modeling and a large set of parameters that are unique within the space systems domain. Furthermore, ESA-AF supports the representation of programmatic and procurement activities, which are central in the European space context.
- **Easy to use by a large number of different stakeholders:** Space programs are multinational undertakings that involve a large number of actors with very different backgrounds regarding domain experience, their roles (politicians, officials, managers, researchers, spacecraft engineers, software engineers, and various other technicians), technical skills, and their culture. Successful communication on various abstraction levels is a key for the success of the whole program. The ESA-AF addresses this need and supports SoSE by maximizing the effectiveness and alignment of technical and strategic decisions.

ESA-AF was used to support the SoSE activities for GALILEO (the upcoming European Global Navigation Satellite System), Copernicus (formerly GMES, the Global Monitoring for Environment and Security), and the Space Situational Awareness (SSA) program, which is a system that warns about dangerous situations in the outer space.

19.3.9 OMG Unified Architecture Framework® (UAF®)

Brief description	
First year of publication:	2017
Developed resp. published by:	The Object Management Group® (OMG)
Primary field of application:	Broad range of complex System of Systems
Development process or methodology:	No
Meta model (Ontology):	UAF Domain Metamodel (DMM)

The great variety of architecture frameworks, especially in the defense industry, offers not only advantages. Many similarities, but also a lot of differences in DoDAF, MODAF, NAF, and also other defense-specific frameworks made it difficult to develop an integrated and consistent architecture while involving all the stakeholders of different nations and forces. Thus, various concepts in the different frameworks have the same name and also the same stereotype, but different definitions, i.e. different semantics, which leads to misunderstandings and an inconsistent architecture when used internationally. Also modeling tool vendors are challenged to support a variety of architecture frameworks that have been developed during the last decades to meet the unique needs of several domains and/or nationalities.

Therefore, the Object Management Group (OMG) has made efforts to consolidate especially these frameworks and to develop a multi-purpose architecture framework, that is not only applicable for domains similar to DoDAF, MODAF, and NAF, but also supports a broad range of commercial and industrial applications. The result of this work is the *OMG Unified Architecture Framework®* (UAF®).[2]

UAF in its current version 1.1 (April 2020) defines 73 viewpoints that are organized in 13 domains and 11 view types, sometimes also called "model kinds," in a two-dimensional grid. The freely available specification contains, among others, two normative documents: the *UAF Domain Meta Model* (DMM) [191] and the *UAF Profile* (UAFP) [190]; the latter is an implementation of UAF-specific

2 Unified Architecture Framework® and UAF® are registered trademarks of The Object Management Group®.

concepts using the stereotype mechanism from UML/SysML (see Section A.6 about stereotypes and profiles).

The framework was evolved from the *OMG UPDM*™ (Unified Profile for DoDAF and MODAF), a modeling language standard (stereotype profile) which, as its name suggests, supports the two popular architectural frameworks from the military environment.

The OMG Unified Architecture Framework is based on UML and SysML standards, and therefore, a seamless transition into the modeling of individual constituent systems of a System of Systems compound is possible. As an innovation over previous well-known defense frameworks, UAF also provides a Security View that is aligned to the NIST (National Institute of Standards and Technology) cyber-security goals. This view supports the analysis, specification, and mitigation of security and cyber-security risks.

19.4 System Architecture Framework (SAF)

Together with Michael Leute

	Brief description
First year of publication:	2022 (planned)
Developed resp. published by:	SAF Working Group supported by Gesellschaft für Systems Engineering e.V.; German Chapter of INCOSE
Primary field of application:	Technical System Architecture
Development process or methodology:	Scope INCOSE Technical Processes as defined by INCOSE SE Handbook
Meta model (Ontology):	SAF Metamodel, based on SysML, aligned with INCOSE SE Handbook

19.4.1 SAF and Enterprise Frameworks

Inspired by the various enterprise architecture frameworks, the System Architecture Framework (SAF) is a common, domain-independent architecture framework dedicated to support model-based systems engineering of technical systems. The SAF is a complementary framework to enterprise architecture frameworks supporting the needs of potential system suppliers to enterprise acquires. SAF starts where an enterprise architecture framework stops.

The application of an enterprise architecture framework results often in operational capability needs which may lead to the acquisition of one or more technical systems. The SAF supports the potential supplier of a technical system to

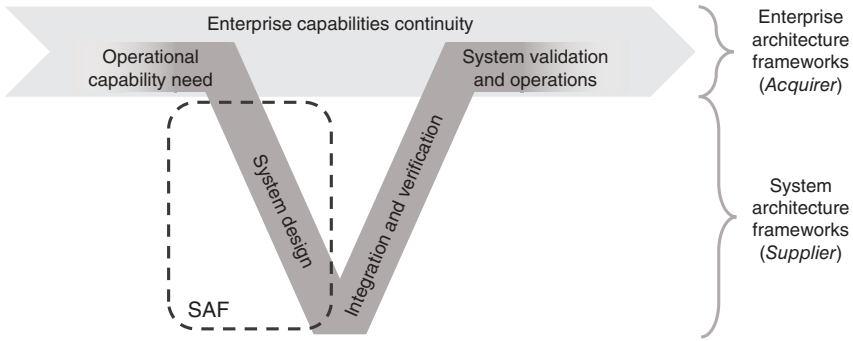


Figure 19.3 Enterprise architecture frameworks and system architecture framework. Source: ©2021 Michael Leute, reproduced with permission.

pick up the baton from where the enterprise framework left it off, as depicted in Figure 19.3.

The SAF defines specific viewpoints to support major aspects of the system design by creating a model capturing the functional, logical, and physical structure [8] of a technical system. For the modeling of the system, behavior, collaboration, and its embodiment in the operational context further viewpoints are defined. The relationship and traceability viewpoints for requirements are established showing their coverage, interface, and domain information model viewpoints are completing the framework [7, 24].

The SAF will be extended to support safety- and security analysis by additional viewpoints that will integrate safety- and security aspects into the system model [71].

The SAF aims to:

- be flexible enough to be tailored to different processes and project scopes,
- provide the appropriate viewpoints for the different project phases with the right level of detail,
- reflect a semantics common to the systems engineering of technical systems supporting communication with interested audiences [115, 265],
- include modeling guidelines and standardized diagram content supporting the systems engineer,
- define stereotypes, naming conventions and model structures to support exchange, automation, model validation, and document generation, and finally
- guide the SAF modeler to choose the relevant viewpoints based on generic concerns.

Due to the generic nature of the underlying SysML language, the SAF does not pose a rigid corset for system architectures, but one that can be adapted and extended as needed: the SAF provides guidance to viewpoint-based tailoring and

extension of the framework to adapt to an organization's specific systems engineering approach and information need. This scalability allows a planned MBSE approach tailored and balanced according to the needs imposed by the circumstances of project and executing organization.

19.4.2 SAF Ontology

The *SAF Conceptual Model* ontology defines relevant systems engineering information items and their relationships.

The *SAF Viewpoint Grid* structures all viewpoints in a grid. The grid rows, the SAF domains, follow the left leg of the systems development life cycle Vee model (see Appendix B). The grid columns, the SAF aspects, group common kinds of viewpoints addressing stakeholder concerns [148].

Each SAF Viewpoint defines exactly which subset of the ontology it contains, which concerns are addressed by the viewpoint, and how the viewpoint shall be implemented using SysML and stereotypes. This vendor-agnostic approach allows to implement SAF in different modeling tools achieving a high degree of interoperability. Currently, the SAF is being implemented for some popular architecture modeling tools.

19.5 What to Do When We Come in Touch With Architecture Frameworks

As systems engineers, we can come in touch with architecture frameworks in different ways.

One possibility is that we need to do the systems engineering for just one constituent system of a System of Systems compound. If this is the case, it may be that we are not even aware about it. For example, in the defense domain it may be that a governmental procurement authority awards a contract for a system development to your organization without that it is obvious for you that the system is intended for an integration into a SoS. The probability is very high that the requirements which are formulated by the contracting authority are derived from an overlying enterprise architecture, resp. from a planned System of Systems. If that is the case, then it would be very valuable to establish a seamless traceability from a possibly existing System of Systems model to the model of your constituent system. This is particularly well possible if both the SoS model, as well as the model of the constituent system are based on the same modeling concepts, i.e. both use the UML or SysML as their core language.

In much rarer cases, a systems engineer comes in the position to develop a System of Systems as part of an enterprise architecture. As we've seen in this

chapter, architecture frameworks are quite different from each other, both in goals and in approach. The huge amount of architecture frameworks available is good news and bad. It is bad, because it increases the difficulty to choose the right one for the current application that suits best. On the other hand, this variety is a great chance, because often the best choice is to blend together different frameworks in a way that works best for the current application. No matter which framework you choose, consider that the major goal of an enterprise architecture is to deliver real business value as quickly as possible to its stakeholders.

20

Cross-cutting Concerns

Cross-cutting concerns are those concerns that are relevant on different levels and layers, but also from different perspectives of the system architecture. There are very different concerns that have this nature. In this chapter, we will discuss some of the major ones. Each of the following sections is about one of them. Finally, trade studies and budgets will be discussed as a means of handling cross-cutting concerns.

20.1 The Game-Winning Nonfunctional Aspects

Fried and Hansson write: "things like speed, simplicity, ease of use and clarity are our focus. Those are timeless desires. People are not going to wake up in ten years and say, 'Man, I wish software was harder to use.'" ([82], p. 85). This statement stresses the importance of nonfunctional aspects, here: speed and ease of use.

The nonfunctional requirements are often cross-cutting concerns, because multiple parts of the system will typically need to be made correctly to satisfy them, but just a minor detail in one of these parts or in one of the systems architecting activities defining them can spoil the success. For example, a noncompliance with requirements about total mass of the system can result from a single part of the system being too heavy. The reaction speed of a system can be spoiled by both a poor layered architecture and an unsatisfactory product architecture.

The system architecture description should describe how nonfunctional requirements are mapped to the different system elements. In the case of requirements about linearly superposing parameters like mass or storage consumption, it may be sufficient to make budgets of maximum parameter values for different system elements (see Section 20.5).

Important nonfunctional aspects are the so-called "-ilities" (which is a term from the "Terms and definitions" section of the INCOSE Systems Engineering Handbook [99]): availability, sustainability, maintainability, manufacturability,

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

reliability, supportability, usability, etc. Even if not ending with "ility," safety and security need to also be on the list of important nonfunctional aspects.

Especially with regard to safety and security, there are changes ongoing due to the continuous growth of information technology and artificial intelligence, which are explained in the following.

Subsystems based on artificial intelligence may not be accessible to theories applicable to deterministic systems, because their behavior may not be determined at design time in case they continue learning at runtime, or because their behavior is so complex that it is not realistically possible to capture it in descriptions that one is used to create of deterministic systems. Therefore, new thinking may be needed in considerations about system safety, in case the system-of-interest is based on artificial intelligence. For example, consider the Virtual Tour system: According to Chapter 2, it navigates autonomously based on artificial intelligence. How will the tour robot be prevented from accidents with pedestrians or other vehicles on site, in case the artificial intelligence fails. Maybe (to just state one of the many alternative approaches for addressing safety in this case), additional safety subsystems with deterministic behavior need to be foreseen to initiate an emergency stop, before such accidents happens.

The more information technology is in the system, the more the cyber-security sub-topic of security becomes relevant. Especially with growing interconnectedness (e.g. in the case of the cyber-physical systems introduced in Chapter 4), the range of imaginable cyber-security threats becomes large and leads to a considerable effort for a reasonable treatment of the cyber-security aspect. While the cyber-security topic has been motivated here based on information technology, its scope is system wide and involves far more than just software technology domains. It may trigger mechanical design considerations (for example: how to mechanically prevent fast access to the internal electrical connections of a user terminal) or other hardware-related topics (for example: how to avoid proceeding with critical operations when the system contains a compromised microchip). Unlike many other activities in system development, the activities handling cyber-security will not stop when the majority of the development activities end. Cyber-security-aware systems engineering will periodically discover the need for changing deployed systems that have already entered their utilization stage. Since new cyber-security threats are discovered continually, the requirements that define a sufficient level of cyber-security can change on a daily basis, with applicability not only to systems under development, but also to systems already taken into utilization.

The cyber-security aspects could be complemented with a long list of literature references. Since even those would probably be partially outdated by the time this book is read, we restrict ourselves to point to a book with cyber-security examples in the model-based context: Borky and Bradley, Effective Model-Based

Systems Engineering [34]. The book (p. 345) mentions the work of cyber-security specialists. These are people who are usually certified in the cyber-security area. Typically, such people are responsible for all cyber-security questions of e.g. a whole organization. We can just recommend to anyone not familiar with the topic to get the help of one of those specialists instead of trying to get insights into this topic via the literature only. The website www.cisecurity.org [1] gives good insights to the range of topics handled by such specialists and by organizations who took their advice.

Due to the cross-cutting nature of nonfunctional aspects, one may need to optimize in different perspectives for satisfying such requirements. For example:

- In an approach based on the functional perspective, one may for example drive the optimization of power consumption by asking: Do we really need this functionality? Can we make it more simple?
- In an approach based on the physical perspective, one can drive power optimizations by asking: Can we optimize the physical implementation of this power-consuming subsystem?

20.2 Human System Interaction and Human Factors Engineering

When we introduced the system context in Section 10.2.2, we placed users like the "virtual tour customer" outside the system boundary. If the user was part of the system, then the user would become just another system element that the system architect can decide to move around in the system or even eliminate from it. In the case of our Virtual Tour example system, this would of course be nonsense. In systems involving operators, it can indeed be an option to move or even suppress the operator. Think of the systems involved in asking questions about your electricity provider's bill by calling this company. In the last millennium, there were times during which one needed to talk to a human operator for establishing the connection between the own home phone and the phone system of the electricity company. Today, we just dial a number and get connected – maybe again with a human operator who works at the electricity company. A trend in this area is to move the operator jobs to countries with low salaries or to spread them across time zones in order to offer services 24 hours a day without having operators do night shifts. Some companies even answer easy questions like "have I paid my last bill" without any human interaction on their side, by means of a computer system that can synthesize voice and asks the caller to navigate by pressing keys on the phone.

Humans can thus be either part of a system or outside the system. In both cases, the human–system interaction has to be considered, and it is recommended

to ensure that the user is getting enough attention, e.g. by ensuring that all human–machine interfaces are explicitly modeled and will be retrieved exhaustively from the model for further processing by human factors engineering experts, usability engineering experts, user experience design experts or whatever your organization calls the corresponding experts.

We should consider all humans inside or in interaction with the system during its full life cycle (so explicitly including, e.g. production and maintenance personnel). Human factors engineering is about fitting the system to these people. This involves, e.g. ensuring that people will be able to use the system properly, but also caring about their safety. It can also include approaches like "design for user experience" Hassenzahl et al. [100], with a focus on the emotions of the people in contact with the system.

Lockett and Powers [159] point out that human factors engineering extends "beyond the application of common sense to design" (p. 493). It is thus a discipline to explicitly account for in work break down structures and plans. Special expertise in this field of engineering is required. It would be beyond the scope of this book to provide more details on this special but very important field.

20.3 Risk Management

Risks are cross-cutting concerns, because the system and each system element down to parts can cause risks.

One has to distinguish product risk from project risk. While *product risks* are potential problems the product can cause during its use, *project risks* are potential problems in reaching the goals of the project, so, for example, potential problems in meeting deadlines, staying in budget or ensuring that the development of the product is feasible at all.

In the scope of systems architecting, both kinds of risk are relevant:

- System requirements can state measures for reducing product risk, which have to be transformed into a solution via systems architecting. Often this activity is intertwined with activities around system safety, because product risks can include risks of harm to humans, for example the actors of the system. Product risks can also be related to security, if a security breach leads to a problem related to the product. A particularly interesting case is a security risk that needs to be considered in safety risk management, for example if an intruder can cause harmful system behavior – a phenomenon that always existed also in classical security considerations, but becomes more and more common when intruders can use more and more information technology vulnerabilities to break into systems via the cyber-space. In this latter case, the safety risk management needs to account for the result of cyber-security risk management.

- During their work, system architects may discover project risks, which they should feed into the project risk management process.
- System architects should participate in proactive activities that aim at monitoring and managing both product and project risks.

The INCOSE Systems Engineering Handbook [265] defines a risk management process that provides the risk management strategy, risk records, and a risk report for a project. During the process, risks are *treated*, this means that actions are planned in case a risk becomes unacceptable. In order to do so, risks need to be analyzed by investigating the likelihood of the occurrence of an undesirable event as well as the consequences in case of occurrence.

Depending on various factors like the kind of products to be developed, the target markets or the kind of organization developing the product, there may be regulations in place that need to be taken into account when doing risk management. We do not go into details on these, since they differ from case to case. We recommend to anyone responsible for risk management to get to know the applicable regulations.

Finally, it has to be stated that looking at risks only may be a too narrow point-of-view. Forsberg et al. [80] write about "opportunities and their risk." They point out that risks arise from pursuing new opportunities and that the awareness of this fact allows for looking at the balance between risks and opportunities and optimizing it.

Seen the importance of risks and opportunities for systems architecting, we are happy to see that it has been proposed for the SysML v2 to enable the representation of risks in the model.

20.4 Trade Studies

Trade studies as aim at making the "right" selection among different alternatives for solving a problem of an engineer, systems engineer or, here specifically, the system architect. They can be considered as a means to help decide in multiple decision situations (e.g. [265]). In the context of the activities described in this book, trade studies are good means, e.g. for choosing the right product architecture during functional-to-physical mapping.

During a trade study, criteria will be defined for deciding on the solution. Then, different solution options will be identified, where some may already be excluded in an early stage due to not meeting requirements toward the system. Afterward, decision-making is applied in order to decide for the solution to pursue. This can be made via a simple pro/contra analysis (which Ullmann [252] calls "Franklin's method" based on a quotation of a letter from Benjamin Franklin in which

step-by-step guidance for a semiformal pro/contra analysis is given). Also, more formal approaches like decision trees (e.g. Skinner [229]) can be considered.

In the end, what counts is to achieve sufficient certainty to select the solution close enough to the optimum. The decision with its related rationale needs also to convince relevant stakeholders. Skinner [229] points out the importance of a communication plan for building trust in the course of actions to be taken during decision-making. Emes [69] recommends to make the selection of stakeholders based on the scope of the decision.

20.5 Budgets

Unlike the budget we give children before they enter a toy store, *budgets* in the following are not about money. This section is about budgeting certain properties of the system per system element in order to be able to meet a nonfunctional requirement whose scope is the complete system.

Budgets are needed if multiple system elements contribute to the same property of the system as whole. Examples of such properties are current consumption, heat dissipation, memory use and mass. One can easily verify that each of these will be made up by contributions of all parts in the system having the corresponding property.

In order to make a budget, one will need to know the relationship between the property values of the system elements and the property value of the complete system. In the example of mass, the relationship is given by the sum operator: the total mass is the sum of masses of all parts. Already when considering storage, the equation may become more complex because overhead for the organization of data structures or the nonlinear effects of data compression algorithms may have to be taken into account.

As soon as one can compute or estimate relationships between the involved values, a budget for each of the parts can be made. For example, the maximum mass of each part can be derived from the specified maximum mass of the system. It is important to pay attention to tolerances when making the budget. Properties like mass are usually specified via a nominal value and a tolerance range. In order to be sure that a maximum mass requirement is met, one has to make a budget that states the maximum value inside the tolerance range instead of the nominal mass. Or more general: budgets are usually based on a worst-case scenario. However, it makes sense in certain cases to exploit statistical effects. In the given example, one can build the system by combining parts with lower than nominal mass and parts with higher than nominal mass such that the deviations from nominal partly compensate for each other.

21

Architecture Assessment

Experienced architects have a certain gut feeling about whether a design is good or bad. Many of the architect's decisions must be made early in the development phase on the basis of assumptions and can, if they have been bad, bring the project to failure. It is often difficult to roll back or revise architecture decisions and their consequences. To put the important architecture decisions on a profound basis, you should do regular assessments of your architectures.

Architecture assessment methods assess the quality of an architecture. It is not an assessment that can be automated and leads to a set of objective performance indicators. Of course, you can support the assessment by scripts or simulations that verify specific aspects of the architecture, but the main part needs the craft of the system architects. It is a balancing act between business goals, technology options, and product management constraints.

The Architecture Trade-off Analysis Method^{SM} (ATAM^{SM})[1] developed by the Software Engineering Institute (SEI) at the Carnegie Mellon University is a common architecture assessment method in software engineering [137]. Since it does not focus on specific software technologies, it could also be applied to the systems engineering discipline (for example, Firesmith et al. [78]).

The standard ISO/IEC/IEEE 42030:2019 defines the term "architecture evaluation" as a judgment about an architecture with respect to specified objectives [111]. ATAM is mentioned in the standard as an example for an architecture evaluation framework.

We recommend ATAM as an architecture assessment method that is based on a structured communication process to incorporate all relevant stakeholders and viewpoints. It focuses on the main important aspects, considers stakeholders to strengthen the decisions, and provides a replicable documentation of the arguments for and against the decisions.

1 Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

ATAM leads to an explicit consideration of the system objectives, related quality attributes, architectural approaches, and decisions.

Typical quality attributes of an architecture are, for example, performance, reliability, availability, security, producibility, disposability, and modifiability (see also [113]). They are tightly coupled to the physical architecture. Some of the quality attributes are also relevant for the functional architecture (Section 17.10). For example, performance attributes about the duration of a function.

The ATAM process can be performed in early phases of the system development project, because you do not need the real system to assess the architecture. It can already be done with early versions of the architecture documentation. That uncovers potential risks and sensitivity points early enough to adapt the architecture if necessary. It also highlights trade-off points that force the stakeholders to prioritize their requirements. For example, to answer the question if it is more important that the system meets the energy consumption or the performance requirements.

ATAM mainly considers three aspects: The goals of the system and the business around the system, the quality requirements that meet the goals, and, finally, the architecture that satisfies the quality requirements. ATAM has nine process steps (Figure 21.1):

1. Present the ATAM process
2. Present the business drivers
3. Present the System Architecture
4. Identify Architecture Approaches
5. Develop a Quality Attribute Utility Tree
6. Analyze the Architecture
7. Identify and Prioritize System Usage Scenarios
8. **Repeat step 6**: Analyze the Architecture decisions based on the outcome of step 7
9. Present the ATAM Result

In the remainder of this chapter, we briefly describe the ATAM process to understand the concepts. We have adapted the ATAM concepts to our architect toolbox and integrated artifacts such as the base, logical, and product architecture into the ATAM process. For more comprehensive documentation of ATAM, please refer to the report from the SEI [137] or the book "Software Architecture in Practice" by Bass et al. [23].

The first three steps are about presenting the ATAM process itself, the goals of the business, and the proposed architecture to selected stakeholders of the system. That clearly shows that architecture is a lot about communication. See also Chapter 23.1 about communication skills for system architects. After the presentation of the ATAM process, typically presented by the lead of the

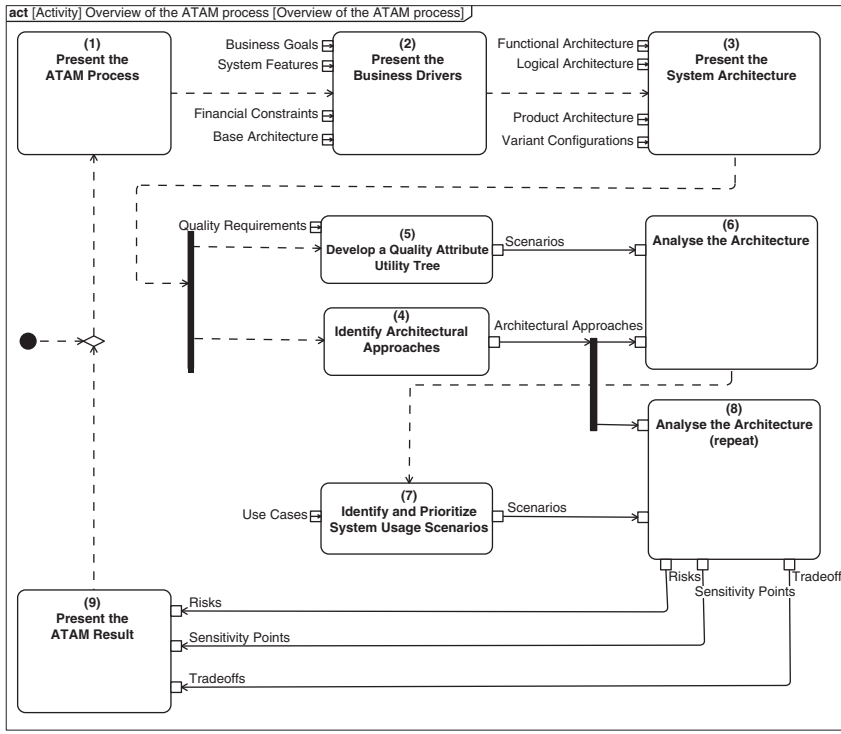


Figure 21.1 Process steps of ATAM.Source: Based on [137].

architecture assessment team, the project manager presents the business drivers of the system. These are the most important features (functional requirements), the business goals, financial and other constraints, and important aspects for the business system context. Figure 21.2 shows the main objectives of our sample Virtual Tour system. These objectives address the concerns of the stakeholders of the system. One of the business goals of the Virtual Tour system is for the vendor to become the market leader in those systems.

Financial constraints are the maximum costs of the development. Other constraints are, for example, the base architecture (Section 9.2), and the organization of the development project (staff, locations, processes). Overall each presentation should not last longer than an hour.

A system architect gives a presentation of the architecture. It should cover the different architecture types, that means the functional, logical, and product architecture, their relationships, and the main elements of the architecture. Additionally, the main driving requirements of the architecture, for example, performance requirements like a maximum mass of the system, are presented, and some main

#	△Id	Name	Text	Stakeholder
1	45	World Wide Access	People from all over the world are able to visit the exhibition.	Exhibition Operator
2	46	Connected Knowledge	The exhibitions' valuable knowledge is linked with other artifacts in the world increasing the overall knowledge.	Exhibition Operator
3	47	More Visitors	The exhibition has 25% more visitors every year for the next 5 years	Exhibition Operator
4	49	Market Leader	We become the world market leader as a manufacturer of virtual exhibition systems.	VT Vendor

Figure 21.2 Virtual Tour system objectives.

use case scenarios are given. The system model is a perfect source for the presentation. Specific diagrams could be created to address the concerns of the audience. All diagrams are views on the same model. The information presented and the information used for the development are consistent, respectively, are the same. Again, that presentation should last round about one hour.

The fourth step identifies the architecture approaches. In the terminology of ATAM, architectural approaches are a set of architecture decisions. An architecture decision can belong to an architectural style. A style provides a vocabulary for the architectural elements and connections and constraints on how they could be connected. It is not a concrete pattern that solves a concrete problem. A pattern can be part of a style. An example of an architectural style in software engineering is a client/server or peer-to-peer architecture. The term architectural style is defined in [85]. Although its origin is the software engineering discipline, it is also valid, usable, and valuable in the systems engineering discipline.

The final step before starting the analysis is to identify and prioritize the most important quality attributes of the system by building a quality attributes utility tree. Quality requirements from the business perspective are often too fuzzy. For example, "The system must consume little energy" or "The system must be easy to use." The utility tree is a tool to break down high-level quality requirements to concrete scenarios. Figure 21.3 shows a utility tree of the Virtual Tour system. The first level is a list of quality requirement categories such as performance, availability, security, and safety. The leaves are concrete scenarios that could be prioritized. Kazman et al. propose a two-dimensional

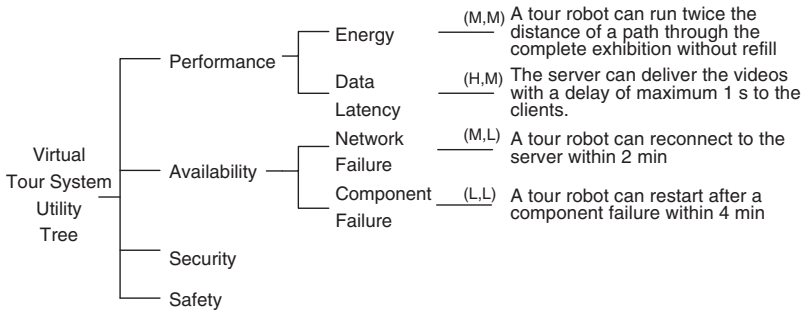

Virtual Tour System Utility Tree
Performance
Availability
Security
Safety
Energy
Data Latency
Network Failure
Component Failure
(M,M)
A tour robot can run twice the distance of a path through the complete exhibition without refill
(H,M)
The server can deliver the videos with a delay of maximum 1 s to the clients.
(M,L)
A tour robot can reconnect to the server within 2 min
(L,L)
A tour robot can restart after a component failure within 4 min

Figure 21.3 Excerpt from the quality attributes utility tree for the irtual Tour System.

prioritization [137] where the first dimension states how important the success of the scenario is for the system and the second dimension states the criticality of the successful achievement of the scenario. Simple stages for a priority are High (H), Medium (M), and Low (L). During this step, you typically identify new requirements or requirements that need to be changed. Therefore, you should involve the requirements engineers at this point. It is another example of the close collaboration of requirements engineers and system architects. See also Sections 9.1 and 12.2.

The central step of the ATAM process is the sixth (and eighth) step. It analyzes the architecture decisions – identified in step 4 – and reports about the risks, sensitivity points, and trade-offs of each decision. The utility tree – developed in step 5 – and the included prioritized scenarios are the entry point of the analysis. Starting with the high prioritized scenarios each scenario is linked with the architectural approaches that are relevant to achieve the scenario. Output of step 5 is a list of the architectural approaches together with a list of risks, sensitivity points, trade-offs, and questions. Usually, the quality scenarios have an impact on physical architectures and less on functional architectures.

The utility tree is driven by the quality requirements of the system. In the seventh step of ATAM, we brainstorm scenarios that describe the usage of the system and scenarios that describe potential changes of the system. A good source for the usage scenarios is the list of use cases identified during the requirements analysis of the system (Section 10.2.3). If you identify further usage scenarios, You have probably found a new use case and must integrate the new information into the requirements and use case analysis. The change scenarios are used to test the capability of the system for future changes. Kazman et al. describe two kinds of change scenarios: growth scenarios and exploratory scenarios [137]. A growth scenario describes expected changes in the near future, for example, an increased number of virtual tour customers who uses the system, or to combine Virtual Tour systems to provide a system across several exhibitions to a huge global virtual exhibition. An exploratory scenario is an extreme growth scenario. An example of such a scenario could be that the number of simultaneous users of the system grows exorbitantly. They are not expected to occur, but help to identify sensitivity points of the architecture.

The eighth step repeats the analysis step 6 with the outcome of step 7. Finally, step 9 is a presentation of the overall outcome of ATAM to the stakeholders. Of course, the outcome of ATAM could result in changes of the architectural decisions, quality requirements, and even the business goals. After applying those changes, you can restart the ATAM process, and so on.

We recommend not to consider the ATAM process as a strict workflow instruction. Keep the principles of ATAM and adapt the process to your specific needs.

22

Making It Work in the Organization

22.1 Overview

None of the authors has so far seen an organization that was founded with a system architecture department in place. Most organizations discover over time that development of their products or other systems of their interest implicitly goes through systems architecting processes, which should be made more explicit – for example, for the sake of efficiency and quality or to be competitive. Introducing explicit systems architecting roles and responsibilities in an organization is an organizational change process, and it is not an easy one: While some organizational changes aim at making the same work more efficient, the introduction of explicit systems architecting in an organization may lead to promoting a different kind of work, which requires a different kind of thinking and acting.

Even in case systems architecting is well established in an organization, it has to be kept operational during future organizational changes. A continuous organizational rollout of the methods and the mindset of systems architecting is required from those who are convinced that systems architecting is one of the enablers for success – no matter how much has been established before. In organizations that are used to systems architecting, this continuous process ensures a reminder for most and an introduction for new employees. In organizations that have not been doing it before, the process ensures that the organization learns to explicitly architect its systems-of-interest.

No matter whether systems architecting comes top-down from management or bottom-up from engineers who like to establish a new mindset: There needs to be continuous efforts to make and keep systems architecting work in the organization. In this chapter, we first discuss the organizational structure around systems architecting and then provide some recipes from our own experience for making systems architecting work in the organization. One can usually not assume that the people driving system architecture in an organization are in the position to perform an organizational change project. This chapter therefore does not discuss the

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

management of an organizational change. It focuses on approaches that can help creating buy-in for systems architecting on a peer-to-peer level. Even if you are currently driving an organizational change, you will still need to do this, in order to make the stakeholders enthusiastic about the value of systems architecting.

22.2 Organizational Structure for Systems Architecting

The *organizational structure* or *organization structure* means the "formal patterns of how people and jobs are grouped in an organization" (definition according to the glossary in the book "Organizations" by Gibson et al. [92]). We can represent it with an organizational chart. Blanchard [28] presents different organizational charts for showing potential organizational structures that support systems engineering via dedicated entities in the organization.

Since our scope is limited to systems architecting, the organizational charts we discuss have their focus on systems architecting. One should consider grouping different systems engineering disciplines into one systems engineering entity, which is, however, not shown here because it would be beyond this book's scope. In other words, where we write "systems architecting" one can as well read "systems engineering" and imagine "systems architecting" as one entity inside it.

Inspired by Blanchard's mentioned organizational charts, we show some hypothetical organizational structures that host systems architecting work in Figure 22.1. The names of the different levels and of some of the organizational units are taken over from Blanchard [28]. We expect these names and the whole shape of the organizational chart to vary from organization to organization, and they should therefore be considered as examples only. We have chosen examples that represent the traditional hierarchically divided kind of an organizational structure, because we assume that it is well known. In reality, we may find different kinds of organizational structures. Some examples are summarized in Gray's and Vander Wal's book "The Connected Company" [93].

All three different organizational charts in Figure 22.1 aim at ensuring that the system architect role we defined in Chapter 13 is assigned to some people working inside the organization:

- In Figure 22.1a, the role is assigned freely to employees with the appropriate skills, where needed. These employees should of course be trained as system architects (see Section 13.6) and should be organized as a system architecture team (see Section 13.3), which is not shown in the organizational chart. The people with the system architect role form a community of practice [272].

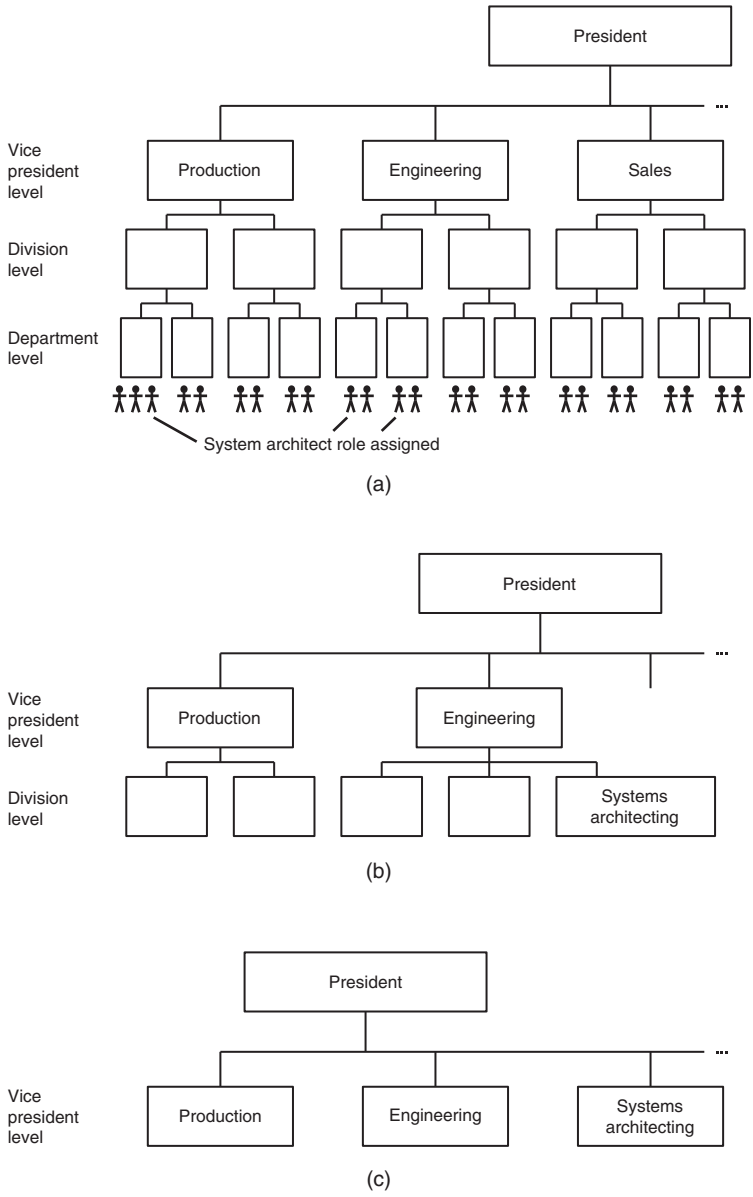


Figure 22.1 Examples showing different alternatives for anchoring systems architecting in the organization: (a) assignment to selected employees; (b) anchoring as a division of its own; (c) anchoring as an entity on the highest possible level.

- In Figure 22.1b, there is a dedicated organizational entity for systems architecting. It resides inside the entity "Engineering."
- In Figure 22.1c, a dedicated entity for systems architecting directly reports to the president. It is there as a hypothetical possibility to be used in the discussion of the advantages and disadvantages which will follow.

In order to find the optimum anchoring of systems architecting in the organization, the different options from Figure 22.1 have to be compared with each other. This is what we will do in the following, but first we like to point out that also mixtures between the different options need to be considered in order to increase the likelihood that a good solution is found. We will first discuss the which level in the organization to choose for systems architecting before we compare the pure role approach according to Figure 22.1a with the ones based on organizational entities according to Figure 22.1b,c. In the end, we will discuss mixtures between the different options.

When we consider that the system architect has to optimize the system in a holistic way, then an organizational entity that is responsible for systems architecting should in a straightforward approach be placed on an equal level or on top of those entities that define and implement the different subsystems. Based on this consideration only, placing systems architecting on the same level as the different engineering disciplines like shown in Figure 22.1b is probably a good option. However, let us revisit a finding from Chapter 12: The system architect should not only be concerned with the system-of-interest but also with its enabling systems. Consider production systems like automata used during assembly, programming, and production testing of the final products as one example of enabling systems. In the organizational charts in Figure 22.1, they may be in the responsibility of the entity "Production," so outside the "Engineering" divisions. In organizations in which interactions are difficult across high distances in the organizational chart, one would thus need to place systems architecting at least on the same level as the "Production" entity, which would result in the variant shown in Figure 22.1c. The "Production" entity was just an example here. There are many more such entities with relevance for the system architect that can only be reached directly on the level shown in Figure 22.1c. Examples are the information technology entity and the one(s) ensuring maintenance and after sales support of the products.

We can now ask ourselves if a systems architecting entity directly reporting to the president like shown in Figure 22.1c is reasonable. In many cases, it is probably not. In organizations in which interactions are difficult across high distances in the organizational chart, the system architects might need to be placed on a level where they are close to the very important engineering stakeholders, especially developers doing the actual implementation work. In organizations in which hierarchical distances do not matter, the position of a system architecture entity does not matter so much anyway.

It should be questioned whether at all systems architecting needs to be represented by an organizational entity or whether it can be implemented by assigning the system architect role to different people in existing entities, like shown in Figure 22.1a. Before discussing this, let us revisit this role assignment approach and add some details to it: The idea is to have people with an engineering background and existing assignments inside the engineering disciplines of the organization do systems architecting. They can be trained as systems architects. They then take the system architect role and use parts of their work time for systems architecting work. The rest of the time remains available for their other engineering work. The people with the system architect role should be joined in a system architecture team, and the leader of this team has to be defined. The clear advantage of this approach is the direct link it creates between the engineering work and the systems architecting work. It ensures that system architects are aware of the current situation in the engineering disciplines and that they can easily communicate the system architecture to the engineering disciplines in a peer-to-peer fashion. Of course there are also some disadvantages to the approach. Here is a comparison:

- Advantages of assigning systems architecting to individuals in the organization via the system architect role according to Figure 22.1a:
  - The systems architects are in close contact with the engineering disciplines.
    - This eases the maintenance of communication and networking between systems architecting and engineering.
    - The system architects stay in touch with reality. In this context, it is noteworthy that Muller [178] sees system architects "drift away from reality" as a consequence of working in abstractions too long.
    - The system architects stay up-to-date with new evolutions in the different fields of engineering.
    - The system architects maintain credibility in the engineering disciplines because they are recognized as peers.
  - The organizational structure does not need to be changed. This is an advantage particularly if systems engineering is new for the organization, which implies that there is no knowledge or experience with systems architecting that would allow for deriving the appropriate organizational structure.
  - The efficiency of systems architecting is more invariant to organizational change than in the other shown options because the system architects are used to work in a network across the organization.
- Advantages of creating a dedicated systems architecting entity in the organization according to Figure 22.1b,c:
  - There is visibility of the systems architecting tasks in the organization.
  - It is easy to understand who has to handle the systems architecting tasks.

  - Workload in systems architecting can be made more predictable over time (see also: Section 12.6).
  - The people in the systems architecting entity can fill their work days with systems architecting tasks, without conflicts with engineering tasks
  - A person in the systems architecting entity can become an expert in systems architecting without a conflict with the need to stay expert in an engineering discipline.

The last two subitems of option (b) and (c) can also be the case with option (a), if the people with the system architect role are allowed to use their time fully in that role.

The above comparison shows that there is not the ideal solution. One needs to optimize the setup based on the current situation in the organization. The optimum setup may be different after reaching a certain maturity in systems architecting than during the first time an organization consciously does systems architecting.

As mentioned above, it is also possible to mix the different options:

- One full-time architect or a few ones can work in a dedicated entity in the organization and run architecture teamwork with individuals from the engineering disciplines to whom the system architect role has been assigned.
- A dedicated entity according to Figure 22.1b can host the system architects that work on the system-of-interest, whereas several individuals who work on enabling systems get the system architect role.

The conclusion is thus that there is not the ideal organizational structure anchoring systems architecting in an organization. The organization needs to be designed individually, based on the specifics of the business, the enabling systems, the maturity of systems architecting in the organization, and many more factors. No matter whether there is a dedicated entity for systems architecting it seems to be a good idea to have direct links to people inside the entities of the organization that are concerned with engineering the system-of-interest and developing or purchasing and operating the enabling systems. Whether this is achieved by assigning the system architect role to people inside those entities or by establishing a good network for the systems architects has to be determined on a case-by-case basis.

22.3 Recipes from the Authors' Experience

The following recipes aim at making systems architecting work in an organization that has not consciously done systems architecting before on the one hand, and on the other hand, they should help maintaining and improving systems architecting in organizations in which it has already reached a certain maturity.

22.3.1 Be Humble

We have seen that there are many stakeholders of system architecture (Chapter 12). With the right mixture of luck and professional skill, some of them will enable no less than the success of the system: The requirements engineers will have described the right product for being successful, the engineering disciplines will have properly implemented the subsystems, and the verification people will have observed everything that needs to be observed before a release to the market, well before the release.

And the system architects? Are they contributing to the success of the system? Of course. A system with a really bad architecture is not likely to have success, whereas there are ways in which a good architecture contributes to success (as it has been discussed in Chapter 3). It is thus allowed for the system architect to be proud of having worked on successful systems. However, the system architect should be humble and acknowledge the contribution of the stakeholders as the ones that had the direct impact on the system's success. Otherwise, the system architects might be perceived as parasites who are living on other people's successes. This may be an impediment to the commitment to systems architecting in the organization.

22.3.2 Appraise the Stakeholders

In order to be able to stay humble as recommended before, the system architect may consider that the stakeholders of the system architecture are the real heroes, whereas the system architect is the one coordinating between their disciplines and ensuring that the solution is holistically seen the right one. The system architect should show the stakeholders that he is aware of the importance of their contribution. On this basis, it is easier to show the importance of the system architect's own contribution to successful system realization. Ideally, the system architect together with the stakeholders will be seen as the winning team whose member can only reach success efficiently if working together as a team.

22.3.3 Care About Organizational Interfaces

Important organizational interfaces for the system architect are the ones to the architecture stakeholders. The system architect should periodically assess whether these interfaces are well established, in other words: that the network is alive. This is independent from the approach of anchoring systems architecting in the organization, so it is applicable to both people working in a dedicated organizational entity for systems architecting and people to whom the system architect role has been assigned.

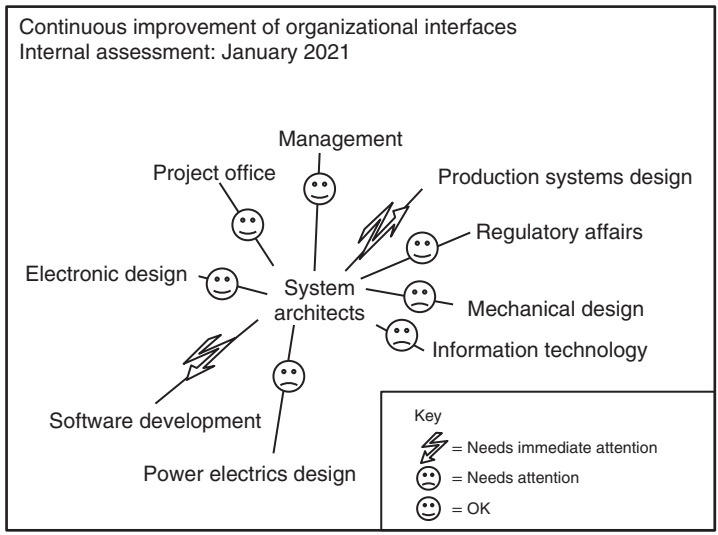


Figure 22.2 Example showing the rating of how well organizational interfaces are established. The organizational information is freely invented.

The periodic assessment of organizational interfaces can be carried out according to the following procedure:

- Consider all possible entities you know in and around your organization and rate whether the system architects' interface to that entity is important
- For the important interfaces from the previous step: rate how well the interface is established.
- Where there is a mismatch between importance and strength of the interface, there is need to act. Assess whether the need is immediate and then create a map like shown in Figure 22.2.

When there is a need to act, define a strategy for improving the organizational interface. Here is a non-exhaustive list of proposals how this may be achieved:

- Prioritize systems architecting work for activities involving the stakeholders with whom the interface needs to be improved.
- Invite the given stakeholders for sessions of the system architecture team and ensure that the agenda items to cover during these sessions are relevant for them.
- Extend the scope of the tasks to be done for the given stakeholders: ensure that more detailed work than usually is done in order to reach a better common understanding with the stakeholders.

22.3.4 Show that it Was Always There

As shown in Chapter 5, each system has a system architecture. This is true even if there has never been a dedicated systems architecting task during the realization of the system. The system architecture may thus have been established implicitly while developing the system. When stakeholders had to learn cooperation with system architects, then we often noticed they were concerned about additional workload. In this case, it may be helpful to explain that the stakeholders themselves are the ones implicitly doing the systems architecting tasks in a world without system architects. The systems architecting work can then be recognized as something that has always been done. The difference with proper systems architecting in place is that the corresponding work is done explicitly and that each system architect helps getting it done. Instead of imposing additional workload, a system architect thus helps finishing the work.

22.3.5 Lead by Good Example

It is own behavior that will drive change. This is also the case in establishing and sustaining systems architecting in an organization. The system architects should comply with their own paradigms if others are expected to follow them.

For example, if it has been decided to follow the model-based approach and to make the model the single source of truth, then the system architect should never archive a temporary document "just for now until we have time to put it into the model." Of course, we have seen in Section 11.11.2 that it may be possible to work on printed posters for the sake of conducting efficient workshops and to feed information back into the model later. It has also been pointed out by Scott Ambler [18] that some models do not even need to go to the archives. We would not like to encourage anyone to start pointless activities just because the direction they follow may have some superficial resemblance of a given paradigm. We rather encourage you to look for the good examples that show the value of chosen way of systems architecting. In the given example, one may say: "in this workshop, we have made a lot of modifications of the model on a poster. We will put them in the model later. However one key result of this workshop is to add a new operation to the Museum Robot's service interface. We will now open the modeling tool and insert it together so that it becomes visible for all model users right now." If it then turns out that the modeling tool is too cumbersome to use in front of a group of people, then the homework has not been done.

In order to lead by the good example, one needs to do homework. If you are *uncomfortable* with the methods, processes or tools that you are promoting, then you should assume that they will be *unusable* for others. So, better come prepared and make things work well for yourself before you ask anyone else to follow you.

22.3.6 Collect Success Stories and Share them When Appropriate

Systems architecting generates value indirectly: It enables the organization to achieve certain benefits, like it has been discussed in Chapter 3. Therefore, it is necessary to show how systems architecting contributed to the success of the organization. This can be achieved by telling success stories about systems architecting at the appropriate moment. One of the authors has just recently used a four-year-old success story to get the budget for a new systems architecting activity approved, because the old success story matched the current situation so well that it became hot news again.

We have experienced that it is problematic to use stories like "Company XYZ published a report proving that they have doubled productivity through systems architecting." No one can prove that the preconditions for company XYZ were the same as they are in your organization and in your current situation. It is much more appealing to use success stories from the own organization, because people will remember the case and will be able to confirm the story intuitively, even if the data backing it up is poor compared to published reports. But the only way to have these stories ready for telling them when needed is to collect them when success is achieved. Build your own collection of success stories and remember to update it on each new success.

What can go into these success stories? In Section 3.3, we already recommended to collect affirmative feedback. Such feedback can be archived and can be used later to tell a success story. But we strongly recommend to ask feedback providers for permission before quoting them in highly visible documentation. Apart from affirmative feedback, you can use your own perception of successes as long as they are transparent for others. In case of doubt, have your success story reviewed by a stakeholder.

Success stories can be made as case reports: what was the problem, how was it solved and how did the chosen systems architecting approach contribute to success considerably? Avoid collecting success stories that start with a problem like "The XYZ department had forgotten to analyze a dependency" and then tells the story of the heroes from systems architecting that saved the boat. Some people will not like the story (especially the ones from the XYZ department), and this may spoil the whole message. Good success stories are in a shape like "A typical system XYZ project takes N months, but in project ABC we needed two months less. We had applied the Functional Architecture for Systems (FAS) method, triggered by the problem that a new requirement needed to be taken into account, without any prior knowledge in the organization on how to satisfy it. We chose the FAS method, because the problem space and the solution space needed to be analyzed in a structured way."

Ideally, you have a template ready for collecting success stories. One of the authors uses an empty slide-show in the organization's corporate design. The template consists of a title "Success story about XYZ" and some instructions. The writer is supposed to fill in "XYZ," to produce a visually appealing case report that tells the success story, and to continue the story in case the amplitude of the success or the evidence for the contribution of systems architecting to success increases over time. It is important to record the preconditions and the achievements. This makes future situations comparable with the one at hand. As a consequence, it can be judged in future whether learnings from a success story are applicable to a new situation.

Tell success stories when the situation is appropriate. This may be when you are asked to report status or when you like to get approval for a new activity or a budget. On the occasion of a status report, you may state the usual "activity XYZ finished within schedule and within budget" and then continue stating "but this time we got the feedback from the engineer John K. that he had a much better overview of his task right from the start of the activity, because the system architect had generated a special view from the model to visualize the context and the fundamental assumptions in an intuitive diagram." When asking for budget, you might state "Last time we applied this method we finished 2 months earlier than usual. This time we would like to invest into some tool support for the method."

22.3.7 Acknowledge that Infections Beat Dictated Rollout

Herrero [101] points out that changes can travel via networks and "infect" others. It is thus important to do networking in order to make changes toward better systems architecting happen.

System architects should not believe that the word "system architect" on their business card automatically gives them the authority to just dictate the valid system architecture. The definition of the system architecture results from collaboration with the stakeholders rather than from a one-way rollout.

If it is possible to convince the stakeholders of the value of systems architecting while working with them, then the chances are high that commitment to systems architecting will spread like a virus. Of course the system architect ideally needs the last word if the system as a whole would suffer from a solution that is proposed by one engineering discipline only. However, instead of optimizing solutions by means of having the last word, the system architect can use stakeholder-specific views to make stakeholders aware about the consequences of their work on the system as a whole. When the stakeholders understand the impact, it is usually not necessary anymore to convince them of a certain holistic solution. They will see the reasoning for it themselves and commit to it much more easily.

22.3.8 Assign the System Architect Role to Yourself

What to do if the organization is not at all ready for systems architecting and you think you are the only one realizing that it is necessary? In Chapter 13, we have seen that a system architect role can be assigned to an individual like a president role can be assigned to an actor who has never been elected. So why not assign the system architect role to yourself even if there is no role description for it in your organization. Of course you should not stand up and proclaim "I have made myself a system architect." You should rather use inspiration from the systems architecting approaches that you can for example read in this book and find out how you can make your daily work more efficient by using them. Before trying to spread anything in the organization, you should first prove to yourself what works well in your work area and what does not work. Once you have found the right approaches for your own work, your way of working has a good chance of "infecting" [101] others.

You should though be prepared for the day on which you organization decides to officially initiate systems architecting. If you have been pioneering enough good approaches and have achieved enough visibility, then you may have the luck to be involved in the organizational change toward more systems architecting. For the case that you do not have that luck, you should always be prepared that all the approaches you were test-driving as a self-nominated system architect may be overruled by an officially nominated system architect.

If you feel you are in the situation of driving systems architecting as a self-nominated pioneer, then you can seek assistance in the community. Consider becoming a member of the nearest INCOSE chapter (www.incose.org) if you are not yet one. Find out if there are local events maybe even at the place where you live or very close to it. You can use them to meet people who are or have been in the same situation as you. They can give you advice or help you find out that you are not alone with a problem. Eisenring et al. [65] report that some group members of a community working group in systems engineering were surprised how many coincidences they found between each group member's own challenges in daily business and other group members' experiences.

22.3.9 Be a Leader

Systems architects ensure that the definitions in the system architecture and the chosen architectural patterns are being followed. Whoever needs followers is ideally a leader. This is why leadership skills are important for a system architect

(see Section 13.2.5). Leadership should not be confused with stubbornness in insisting on the own approach. Leadership is about behaviors that make others understand why it is necessary to contribute to systems architecting and to comply with the resulting definitions, based on the system architect's understanding of their situation and their point of view. It is about ensuring that it is possible to go into the right direction together while feeling relaxed about it.

23

Soft Skills

The fact that good communication and collaboration are crucial for a successful project is truly nothing new. It is the essence for any successful organization. A prosperous enterprise brings people together and creates an environment of good cooperation and appreciative togetherness. Especially in times of ever-increasing complexity and globalization, humans and their collaboration are often the decisive factor for success or failure.

These are all truisms and known for decades. And also for decades, the creation of an collaborative environment for people is one of the major challenges for any organization. People are different regarding their personality traits, social graces, communication, language, personal habits, education, and friendliness. That means: beyond the occupational requirements of a job, which are often called hard skills, there are other important skills a person should have that are related to the interaction and communication with others. These so-called soft skills are significantly influenced by a person's knowledge of human nature, socialization, feelings, emotions, empathy, personal insights, and cultural factors.

System architects in particular have special challenges in this regard, since their typical activities requires them to communicate with many, very different stakeholders (see also Chapter 12 on architecture stakeholders). They have to understand and clarify requirements, they have to be able to impart concepts and solutions, they have to communicate architecture decisions, they have to mediate and integrate in case of conflicts, they have to coordinate with project management, and much else. Hence, it is not surprising that preparatory courses of the systems engineering certification program SE-ZERT® (https://sezert.de) covers also the topics of communication, conflict management, and social competence.

There are numerous publications and books about soft skills, discussing their importance and suggesting ways how to improve those skills. It is not the purpose of this chapter to cover this topic to its full extend, but it provides an overview about

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

some essential parts. In this chapter, you also will learn about some models from communication psychology and personality typology that are helpful to explain several interpersonal dealings and processes. First of all: These models are all wrong, because they greatly simplify the real complexity of these topics. It is basically not correct to pigeonhole individuals into certain categories. On the other hand, such models are useful to explain certain phenomena in a popular scientific way. Or as the British mathematician George E. P. Box (1919–2013) put it: "Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful" [59].

23.1 It's All About Communication

The internal communication within an organization refers mainly to the communications between members of that organization, in case of a company it is about employee communications. In addition, the internal communication is part of an organization's corporate identity. In this context, a good communication culture should develop a team spirit, a kind of shared identity between the members of the organization.

A vivid internal communication within a company is crucial for its success. This is especially true for development organizations employing knowledge workers. In today's world of ever-increasing complexity and dynamically markets with their unpredictable changes, it is of great significance for the existence of an organization to be able to react immediately on disruptive forces. Dealing with high dynamism, complexity, creativity, and productivity requires a high degree of communication on all levels.

While analyzing communication models, communication scientists basically consider two different categories of communication:

1. Communication as a transport and delivery process of a message between a sender and a receiver.
2. Communication as a social activity within social structures.

These two categories are not to be understood as opposites, because the one includes always the other. The difference is basically in the focus of the communication process. The latter category considered communication not only as the sum of actions of individuals, but as a complex social process with emergent properties.

Basically, communication is fallible (error-prone). No matter how much effort you would put into your communication skills: This error-proneness cannot be fully eliminated. Generally, it is impossible to ascertain after a communication process has taken place whether a comprehension has taken place as intended.

This section highlights the importance of communication in systems engineering projects and discusses some topics that might be useful for the system architect's daily work.

23.1.1 Losses in Communication

The Shannon–Weaver model, named after the American mathematician and electronic engineer Claude Elwood Shannon (1916–2001) and the American mathematician Warren Weaver (1894–1978), describes communication as a point-to-point-transmission of a message from a sender to a receiver [224]. It can be regarded as one of the simplest models and it's general applied in various communication theories. In this model, influencing factors play a important role that were termed by Shannon and Weaver as "Noise." In the Shannon–Weaver model, this "Noise" primarily affects the transmission path between sender and receiver, i.e. it deals with external disturbances only. Therefore, one of the often heard criticisms of this model is that it is too simple to describe the real complexity of human communications.

As with message transfers in telecommunications engineering, communication between people is also never lossless. Just imagine that you should describe the beautiful beach from your last vacation in a way that someone gets an exact imagination about it. No matter how much effort you put into this, your counterpart will never have exactly the same image in mind as you.

The first loss could occur when you try to express your memories of the beach in language. Even if you can remember a lot of details, it is difficult to express every detail in language. "The beach sand was nearly white as snow." is not sufficient to describe the exact color of the sand. The sand color that the receiver of your information imagines depends on many factors. So it may be that she does not know snow and the metaphor therefore did not work. It may also be that your words were ambiguous. While the "nearly as white as snow" might be your own expression for the most beautiful beach, it can also be considered as a complaint about imperfection (just nearly as white, but not really as white as needed to be perfect).

The next loss could occur on the communication path between you (the sender) and your counterpart (the receiver). Just because you said something, it does not have to be arrived at the receiver. At this point, the so-called "Noise" described by Shannon and Weaver comes into play. External influences like disturbances or interferences can affect communications or even prevent it. For instance, there might be issues with the mobile phone network which directly affect the phone communication. However, in human communication, such losses may not always be caused by technical issues. Maybe your communication partner did not pay attention to your words, because she is distracted or not focused on you.

Another loss might occur when your counterpart did not understand you. And that does not necessarily mean that such lack of understanding is caused by language difficulties. Of course, language and translation problems are often the cause of communication problems in international projects. Not infrequently, losses in communications in systems engineering projects are caused by a lack of understanding about domain-specific topics, or caused by disparate domain knowledge between different stakeholders (see Chapter 12). Stakeholders in a system development project have different concerns. Some of them are domain experts, others are users of the system, and others again are business people, but they are usually not experts in systems engineering. Probably one of the greatest challenges for the systems engineer is on the one hand to understand these stakeholders and their needs, on the other hand to explain them how the system of interest will work to satisfy their needs.

But even if there are little or no understanding problems: just because a message was understood and acknowledged by your partner, it does absolutely not mean that she agrees with its contents! Unfortunately, not everyone is aware about this important fact. Many think that it is merely sufficient to post an information to its recipients and everything is fine, but such behavior is, for example, not able to engender a commitment by the recipients of that information. Or to demonstrate it with an example: you too, dear reader, can read this book, but you may not agree with all of its contents. Although we, the authors, would be very pleased to get feedback, you are not obliged to tell us your differing opinions and views.

23.1.2 The Anatomy of a Message

There are a variety of different models about human communication. A well-known model that explains the complexity of communication is the so-called "Anatomy of a Message," sometimes also called "The four sides of a message," developed by Prof. Dr. Friedemann Schulz von Thun [263], a German communications psychologist.

Schulz von Thun postulates that each communication contains always four messages simultaneously and can therefore be received on four levels, sometimes also called channels: the objective content level, the appeal level, the relationship level, and the self-disclosure level (see Figure 23.1).

- **Objective content**: This is the factual level. On this level, we send respectively receive the purely objective information and facts are conveyed. This level is about data, facts, and circumstances about a particular topic.
- **The relationship level**: On this level the message conveys what kind of relationship the sender and the recipient have with one another.
- **The self-disclosure level**: On this level the message conveys information about the sender, something about what is going on in her.

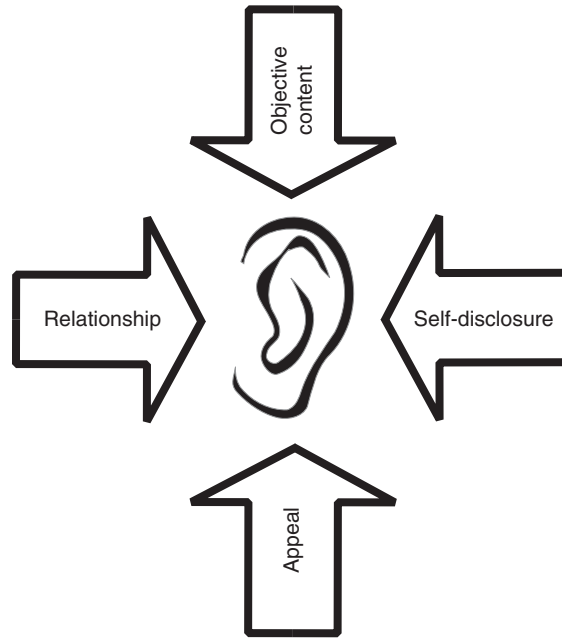


Figure 23.1
A visualization of the four-sides model.

- **The appeal level**: On this level the message conveys a wish, a request, an advice, or an instruction; something of which the sender is hoping that it will have an effect on the recipient.

These four levels of communication exist on both sides, on the sender's side as well as on the receiver's side. Therefore, the probability is relatively high that what was told by the sender is not necessarily what was heard by the receiver. Let us take a look at an example, a communication between two participants in a business context (Figure 23.2).

Of course, it depends very much on the tonality in what intensity each of the four levels of the message are perceived by the receiver. The sender can say it in a factual and sober manner, or with a concerned and worried undertone. Furthermore, especially the relationship level and the self-disclosure level depend significantly on the nonverbal expressions (gestures, mimics, etc.) of the sender. In contrast, the objective content level is simple and evident in this case. Table 23.1 provides possible examples of the four messages contained in the statement shown in Figure 23.2.

By looking at the relationship level, we can discover that there seems to be a disturbance between the sender and the receiver. The sender thinks highly about the receiver, but the receiver hears it as a kind of mistrust on that level. Such disturbances on the relationship level can be critical, because this level dominates the

Figure 23.2 What are the four sides of the message in this communication? Source: © 2015 Jakob K., reproduced with permission.

Table 23.1 The four sides of the message in the communication depicted in Figure 23.2.

	Sender	**Receiver**
Objective content	*The deadline for this document is Friday.*	*The deadline for this document is Friday.*
Relationship	*You know that I have a high opinion of you and your work.*	*She apparently think that I do not stick to our agreements.*
Self-disclosure	*I'm under pressure. I need your help!*	*The customer puts her under pressure.*
Appeal	*You have to have it done until Friday under all circumstances!*	*She wants me to give the highest priority to the document.*

objective content. Paul Watzlawick (1921–2007), who was an Austrian-American psychologist, communications theorist, and radical constructivist, stated that communication has a content and a relationship aspect such that the latter classifies the former and is therefore a meta-communication [127]. That implies that if the relationship level is disturbed, a communication on the factual level is not possible. Therefore, it is important to talk about the disturbance first, and the information must wait until the relationship between sender and receiver is clarified and recovered if necessary.

Knowing about potential misunderstandings in direct communication can be helpful to avoid miscommunication. You can actively avoid that your conversational partner gets angry, offended, or sad. And you may be able to guess why your counterpart reacts surprisingly differently on your message than expected. The knowledge about the anatomy of a message and can remind you about the potentials for miscommunication and thereby keep you conscious to proactively and continually work on good communication.

23.1.3 Factors Influencing Communication

23.1.3.1 The Language

Language affects almost all aspects of everyday life. We need our language to express emotions, share feelings, tell stories, and convey complex information and knowledge. This is also true for communication between all individuals involved in a systems engineering project. Because of language barriers, stakeholders may struggle to communicate what they need or even get necessary information regarding the system of interest. And due to globalization leading to worldwide distributed development teams, language is even more important.

Due to the international nature of many projects and the historical roots of the discipline, English is the dominant language in systems engineering. But it is usually not the mother tongue of all involved people. This could lead to communication problems and misunderstandings. In addition, language is also always part of the culture. We will deepen this issue in Section 23.1.3.4 when we discuss the various connotation of words, and in Section 23.5 when we talk about collaboration skills in an intercultural environment.

23.1.3.2 The Media Used

Ideally, communication should always take place face-to-face, i.e. the sender transmits an information directly to the receiver, typically through language. This is also known as direct communication. But from an economic perspective, this would be too time-consuming and costly for an organization. In a work-related context, it is plain not feasible to exchange every necessary information via direct communication. Thus, the direct communication is often replaced by an indirect communication, also called media communication, which is characterized by the use of different kinds of media. Examples include phones or e-mail.

The indirect communication usually does not have a nonverbal communication level. There is no visual contact, i.e. you cannot see the facial expressions and the gestures, which are also known as body language, of the other communication participants. Exceptions to this are video conferencing systems (see Section 23.1.4).

Furthermore, the feedback is often delayed in this kind of communication. Thus, it can sometimes take days or weeks to get an answer to a question by e-mail.

23.1.3.3 Spatial Distance

Already in his seminal 1977 book "Managing the Flow of Technology" [15], Professor Dr. Thomas J. Allen, an expert for organizational psychology and management at Massachusetts Institute of Technology (MIT), describes a strong negative correlation between spatial distance and communication frequency. The so-called Allen Curve (see Figure 23.3) is a graphical representation that shows this correlation. It is the result of a research project by Prof. Allen in which he explored how the

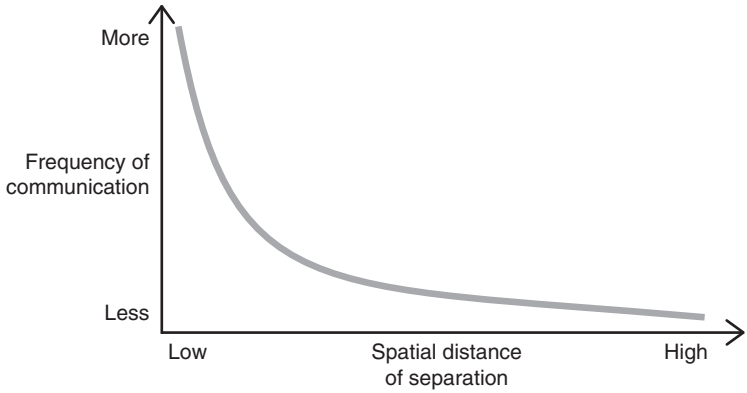


Figure 23.3 The Allen Curve depicts the negative correlation between frequency of communication and spatial distance.

distance between engineers' offices affects the frequency of technical communication between them.

The Allen Curve depicts that we are much more likely communicate with someone who is located in our immediate proximity, e.g. in the same room, as with people that sit further rooms away, on a different floor, or even in a different building. With other words, long distances between members of an organization are detrimental to communication.

Nowadays, we can access a virtual office from everywhere and at any time. The coronavirus pandemic came with good examples showing that distributed working over the Internet can work relatively well. Therefore, one might suspect that today's technologies, like e-mail, cloud-based office systems, or video conferencing systems, are able to bridge the spatial distances and the Allen Curve is not valid anymore. But that's wrong. Recent researches and studies have shown that the Allen Curve is valid not only for direct communication, but also for digital communications. Ben Waber et al. [264] were able to prove that the frequency between colleagues that are working in spatially close distance was four times higher than the frequency with colleagues in distant locations.

In the space industry, it was recognized a few years ago that a close collaboration of engineers is of great importance for the success of a complex mission. Therefore, the European Space Agency (ESA) built a Concurrent Design Facility (CDF) in Noordwijk (Netherland) that allows a team of experts from several disciplines to work concurrently in a highly collaborative environment. The architecture of the building is especially designed to support concurrent working in teams. The CDF consists of several rooms of different sizes and for different purposes. The main design room in the center of the facility, and all other rooms, are equipped with state-of-the-art computer workstations and linked with each

other via a high bandwidth network. All design and auxiliary rooms are not only equipped with video conferencing systems. An audio–visual network allows to display the data from any screen or workstation on any or all of the screens in the other rooms.

23.1.3.4 Various Connotations of Words

Connotation is defined as an implied meaning that is associated with a word in addition to its literal meaning; the latter is also known as denotation. The denotation is neutral and can simply be looked up in an encyclopedia. The implied meaning instead depend upon the social, cultural, and personal experiences of individuals, or it could have emotional reasons. The connotation is an association which the word evokes in a reader's mind, either positive or negative.

The Animal Farm, an allegorical and dystopian novella by English novelist George Orwell (1903–1950), contains many connotations. Many species of animals in the story have a connotation. For example, the pigs connote powerful, corrupt people. "Boxer," a loyal, kind, dedicated, hard working, and respectable workhorse, connotes the working class back then.

Words and terms may have different connotations in different cultures. This circumstance often leads to unintended, sometimes embarrassing misunderstandings. An noticeable example is colors, which have different meanings in different cultures. For instance, the word white, denoting a color, is associated in certain metaphors in European literature with pure, noble, moral, goodness, and innocence. In similar examples, we know from China, by contrast, the color white stands for pale, weak, and without vitality. Therefore, the bride wears a white dress on many wedding photographs, we have seen from Europe and a red dress on photographs we have seen from China.

Furthermore, over time the connotation of words can shift. For example, the word "drug" was originally a neutral term for various active substances, including those that were prescribed by doctors. The term "drug store" was coined in those times. Nowadays, the word "drug" is usually associated with dangerous addictive substances. Hence, in a medical context the words "medicine" or "medication" has replaced "drug."

23.1.4 The Usage of Communication Aids and Tools

In addition to sheer personal conversations, there are many possibilities to facilitate and promote communication within an organization, or tools to support the direct communication. Systems engineers can take great advantages of these possibilities if they are used in the right way.

- **E-mail**: During the last decades, communication via electronic mail (short: e-mail) has not only largely replaced ordinary letter mail. It can also be seen as

an generally accepted alternative to other types of communication, like direct communication or telephone calls.

Nowadays, sometimes e-mail as a media is even considered harmful. The huge amount of e-mails that are received by many employees every day could be overwhelming. Several hundred e-mails a day are no rarity. The processing of the messages is time-consuming and annoying. The permanent sending and receiving of e-mails are setting employees under stress. Some companies have even announced plans to waive e-mail in the medium term.

E-mail should not be used inconsiderately. Before sending an e-mail, you should consider carefully whether it is really necessary. Especially, functions such as "reply to all" or "carbon copy" (cc) can cause unpleasant situations. The message may be, for example, send to not intended addressees. Furthermore, a more economical use of these functions prevents an e-mail flood.

- **Blogs and Wikis**: During the last years, weblogs (short: blogs) and wikis[1] became an important part of knowledge sharing in organizations. Both applications are usually web-based and accessible for members of an organization through the organization's intranet. What makes both applications so powerful for collaboration and communication is its "anyone can edit" capability.

  Due to their chronological nature, blogs are usually used like a diary. For instance, a project blog can be used by members of a systems development project to regularly inform team members and other stakeholders about project progress, important events, new insights, relevant issues, etc. This creates transparency in the project, but the information is usually only accessible for a limited set of users within the organization.

  In contrast to this, wikis behave more like a reference book. They are more suitable for knowledge management for the whole organization, across departments and team boundaries, and beyond end of projects. In some organizations, the wiki is the single source for all company information.

  Wikis and blogs live from participation. So that individuals can get the information out of these applications, they must actively access them. Problems could arise if stakeholders do not actively access the content, because this would lead to a standstill of the information flow via these channels. Therefore, both blogs and wikis allow notification of changes to be sent out via RSS feed.[2] This lets users interested in a specific topic to be notified when changes are made in the wiki or blog. And it could also ensure that the interest about the contents remains high.
- **Instant Messaging (Chat)**: Instant Messaging (IM) provides real-time text transmission over the Inter- or Intranet. It is very similar to Short Message

1 The term Wiki is derived from the Hawaiian word "Wikiwiki" which means "quick."

2 RSS is an abbreviation for Really Simple Syndication and describes a file format that allows to publish frequently updated information.

Service (SMS), a service that allows fixed line or mobile phone devices to exchange short text messages.

The times when Instant Messaging was solely used for personal chat and entertainment are long gone. The use of IM within the business environment is growing at an exponential rate. It provides a powerful possibility to stay in contact with employees, customers, and vendors.

Like any kind of communication via the internet, also the use of Instant Messaging is not without risk. In addition to several security aspects (viruses, worms, malware, intentional, or accidental revelation of confidential material or intellectual property, etc.), it is also not suitable for every communication. It is often more intimate than other electronic media like e-mail, and the communication style is often characterized by internet slang and shortcuts. This does not always fit to a business context, and it can also lead to misunderstandings or sensitivities.

- **Telephone-/videoconferencing**: In times of globalization, worldwide distributed development teams, and a trend toward more home office, telephone- and video conferences are broadly accepted as an alternative to direct communication and can significantly reduce the need to travel. In particular, the coronavirus pandemic has shown that collaboration, sharing, and staying in touch works amazingly well with the help of such tools. Since the availability of high capacity broadband telecommunication services at relatively low costs, video conferencing has undergone a widespread in many domains, such as business, education, engineering, medicine, and media. The range of different video conferencing systems on the market is high. In addition, some social networks also provide video conferencing features.

  One of the strengths of many of these systems is the possibility to share the computer desktop or an application with all participants. This allows, for instance, collaborative work with a system's architecture model.
- **Information radiator**: The term information radiator was first coined by Alistair Cockburn around the year 2000 and is described in his book "Crystal Clear" [45]. This kind of communication aid is very well known in the agile movement. Cockburn describes an information radiator as a well-readable display that is posted in a prominent place where people can see it while they work or walk by. A good information radiator is large, good readable, and can be conceived by observers at a glance. Usually, it is either a board, a poster, or sometimes a screen. It is a kind of one-way-communication and typically used to show status information.

  An example for an electronic information radiator can be found in many software development projects. There, a flat screen on the wall displays a dashboard that visualizes the statuses of the automated continuous build and integration processes.

- **White boards and flipcharts**: Especially in workshop situations, whiteboards, and flipcharts are excellent tools to work together in a highly collaborative way. For instance, system architecture drafts can be sketched and discussed. Questions can be clarified immediately, and all participants could develop a common understanding about the system's architecture.
  The whiteboards or flip charts can be photographed and archived after the session, and the images can be embedded in a blog resp. wiki article. Alternatively, the sketches can be transferred into a SysML modeling tool.
  In Section 17.9, we show the usage of these tools for the collaborative development of a first functional architecture for a system within a workshop.

23.2 Personality Types

All humans are equal, but yet very differently. Since the dawn of time, humans have tried to describe and categorize their personality in many ways. This in turn led to a variety of models, all of them trying to fit something complex and fluid as the human's personality into certain categories. As already noted in the introduction to this chapter above, all of these models are more or less wrong. No matter which personality types model you use, no one on this world is either exactly "type A" or "type B." Like all models, they represent a simplification which in no way is able to represent the complex reality.

Furthermore, it is important to bear in mind that models about personality types are just an aid to predict vaguely how people are likely to behave in certain situations. Those models are inappropriate to determine accurately how a person will behave in any case and under all circumstances. However, what these models can do very well is remind you of that others will not have the same interaction patterns and preferences as you and that it is needed to adapt to others for establishing good collaboration.

23.2.1 Psychological Types by C. G. Jung

One famous model dates back to early twentieth century and was the brainchild of Carl Gustav Jung (1875–1961), often referred to as C. G. Jung, who was a Swiss psychiatrist and psychotherapist. C. G. Jung's model of psychological types was first published 1921 in "Psychologische Typen" [133] (engl.: "Psychological types [134]) and is perhaps the most influential theory in personality typology.

According to C. G. Jung's theory, people can first be characterized by their preference of general attitude. These preferences are **Extroverted (E)** vs. **Introverted**

(I). In psychology, an extroverted person is more concerned with practical realities than with inner thoughts and feelings.

In addition, people can be characterized by their preference of one of the two functions of perception. These are **Sensation (S)** vs. **Intuition (N)**. Sensation means that a person pays more attention to information that comes in through his five senses. In contrast, intuition means that a person pays more attention to the patterns and possibilities that she sees in the received information.

The third pair describes the preference to one of the two functions of judging, i.e. it characterizes how a person like to make decisions. These are **Thinking (T)** vs. **Feeling (F)**. A thinking person puts more attention on objective principles and impersonal facts. A feeling person likes to do whatever will establish or maintain harmony.

The preferences E and I, the perceiving functions S vs. N, and the judging functions T vs. F were combined by C. G. Jung to eight psychological types as shown in Table 23.2.

C. G. Jung stated that we define our psychological type by our so-called *dominant function*.

For instance, if we like to use Extroverted Sensation more than the other functions, then we are an Extroverted Sensation type. We like to gather factual data then, and use our senses to see, feel, touch, smell, and listen what's going on in the world. People who are Extroverted Sensors process life through their experiences and are able to live in the here and now.

In contrast, if our dominant function is Introverted Feeling, then we like to make decisions based on emotions rather than on objective facts and data. People who are Introverted Feelers have the ability "to see through others" and the highest level of empathy of any types.

The practical application of this knowledge is diverse. Knowing about psychological types can be useful for career planning, communication, education, coaching, and counseling. It can improve our awareness of understanding of reactions of other people to situations.

Table 23.2 The eight psychological types by C. G. Jung.

	Perception	**Judging**
Objective	Extroverted sensation	Extroverted thinking
	Introverted sensation	Introverted thinking
Subjective	Extroverted intuition	Extroverted feeling
	Introverted intuition	Introverted feeling

23.2.2 The 4MAT System by Bernice McCarthy

In the role of a system architect, you are frequently in the situation to explain stakeholders the system's architecture and to justify your architecture decisions. This is usually done in a presentation. For your audience is your presentation in most cases a learning situation, as they have to learn and understand new things. Individuals differ in how they learn, they have different learning styles. This fact has been recognized and scientifically studied by the American educational theorist David A. Kolb in the 1970s. Kolb developed a model about different learning styles and published it in 1984 [143]. The model served as the foundation for Kolb's experiential learning theory (ELT). Kolb is renowned in educational circles for his assessment tool named Learning Style Inventory (LSI).

Based on David Kolb's learning styles model, the American educational scientist Dr. Bernice McCarthy developed the 4MAT System [168]. McCarthy has figured out that people ask themselves at least one of the following four basic questions while they are learning:

- **Why? (motivation/the philosopher)**: Typical questions: Why should I learn this? Is it beneficial for me?
  This type of learner scrutinizes the sense and wants to know why your presentation about system architecture should be of interest to him.
- **What? (knowledge/the scientist)**: Typical questions: What are the facts? Can I have more details, please?
  This type of learner wants facts, data, and would like to get an explanation of the thing (e.g. properties of the system's architecture).
- **How? (demonstration/the practitioner)**: Typical questions: How does it work? Can you please show it to me?
  This type of learner wants to get deep knowledge about how something works and would like to try immediately everything out (e.g. build and test the system according to your architecture).
- **So what? (debriefing/the visionary)**: Typical questions: Where can I apply it? What if …? Is it possible to …?
  This type of learner seeks for more, explores hidden possibilities, and thinks about future scenarios (e.g. extensions of the system's architecture)

Knowing about these four types can be a great advantage while preparing and performing your presentation. With this knowledge, you can structure your presentation so that you can satisfy the needs of all four types of learners. You should start with a Why-part to get those ones of your audience who demand applications and examples. In the next part, you should provide many facts about your architecture for What-people. Then, you should provide details about how it works for the How-people, for instance, you can start a simulation in a modeling tool that

demonstrates how the system will work. Finally, you should provide an outlook for the So-What-people, such as future extension possibilities of the architecture.

A further advantage of this knowledge is that we can avoid misunderstandings. If communication problems occur, I can try to classify my counterpart in one of the four categories and adjust myself accordingly. Otherwise, it could happen that What-people and How-people can come into conflict.

23.3 Team Dynamics

As already noted in the Section 23.1 on communication, system architects will need to work across multiple disciplines of engineering. Hence, they may need to work with stakeholders from multiple departments of an organization. This can expose the system architects to situations in which a group of stakeholders is gathered for the first time around a task (for example: a trade study) that is driven by the system architects themselves.

When people need to work as a team, without having had to work in the given group before, the phenomena of team dynamics will become apparent and may become the dominating challenge in accomplishing the task. Therefore, system architects should be aware of team dynamics. To understand the related phenomena, one model has gained significance over the years, explaining the different stages of team development [33]: The so-called "Tuckman model" after Bruce W. Tuckman [250] postulates that team development will always follow a path through the following stages:

- **Forming**: Testing the ground rules for behaviors in the team.
- **Storming**: Inter-group conflicts, lack of unity around interpersonal issues.
- **Norming**: The team develops cohesion. Roles and norms are established.
- **Performing**: The team is a very efficient problem-solver and channels its energy into the given task.

Later, also the "Adjourning" stage [251] was added, considering that teams will not only be created, but also dissolve at some point.

The leader of a team will need to facilitate the transition through the different stages of team development. Experience shows that failure to do so will "heaten up" the storming phase, which will make the team less efficient. Such failures to accommodate proper team development are typically the result of time pressure, which leads to the paradoxical phenomenon that the wish to save time will lead to waste of time in unnecessary "storming."

We recommend to leaders of newly established teams to plan for enough time to go through team development together. Our experience shows that the journey through the stages of the Tuckman model cannot be avoided, but it can

be accelerated if the leader helps the team to cope with the phenomena of team dynamics. The leader can reassure the team members that certain initial effects, like the need to find ground rules, are normal and need to be accepted. The leader can also facilitate the transition through the first stages of the Tuckman models by facilitating an explicit discussions about roles and responsibilities in a newly established team. In the case considered here, the system architect may be the person having the lead to establish a certain team and to take this facilitating role.

23.4 Diversity and Psychological Safety

Building an inclusive corporate culture in which diversity can thrive is currently on the minds of all organizations. Diversity is considered to be one of the key success factors in meeting the corporate challenges of the future. Teams made up of people of different genders, backgrounds, languages, sexual orientations, qualifications, ages, etc., are said to be particularly creative, innovative, and efficient. Studies by international management and strategy consultancies, such as McKinsey & Company [57], conclude that companies are more successful the more diverse they are. However, such studies often have a study design that is specifically geared to executives in upper management hierarchy levels, and thus often only focus on the effect of diversity on boardrooms. They are therefore only of limited value in terms of the effect of diverse teams in other areas of the company.

Diversity requires certain framework conditions under which it shows positive effects. Nowadays, there is much evidence that psychological safety is the essential foundation.

23.4.1 Project Aristotle (Google)

The concept of psychological safety was founded in 1999 by Amy C. Edmondson [63], Professor of Leadership and Management at Harvard Business School, and has been taken-up and verified since then in various research projects (e.g. on innovation, agile, and diversity). But the concept became broadly known much later, in 2016 through Google's research project named "Aristotle" [273]. In 2012, Google launched an initiative to study 180 Google teams over two years and to find pointers that show why some fail while others achieve best-in-class performance. The underlying hypothesis was that, above all, the composition of a team must be of decisive importance. In addition, different leadership styles, personality types, and working methods were the focus of the data collection.

However, it became clear relatively quickly during this study that none of these mentioned factors were really fundamentally decisive for a team's performance.

Instead, "Project Aristotle" found a few new perspectives of relevance that were not expected. For example, it was found that members of a team care about the meaningfulness of their work, and also that their work also ties into the meaning and mission of the organization. Furthermore, it was also relevant that a team's activity has a significant effect, i.e. that its work really makes a difference.

Above all, it turned out that psychological safety had a particularly outstanding effect. How much do the individual team members feel they belong? Do they trust each other enough to open up? Do they trust their leaders? Do they believe that their individual perspectives and values are taken seriously?

If there is no shared belief among all members of a team that there is safety within the group to take interpersonal risks, then everything else comes to nothing. In an atmosphere of fear, even the most diverse team will not perform well. A lack of psychological safety leads to poor decision making, low morale, an increased stress level, and finally a lack of innovation.

23.4.2 Elements of Psychological Safety

So what are the elements that provide psychological safety and create an atmosphere of fearless and inclusiveness? Here are the five essentials:

- **You have to be able to express your own professional opinions openly**: Each member of a team can contribute ideas and question the decisions of others without risk; differing opinions are respected and discussed; there is trust that others will not try to undermine you.
- **Everyone speaks equally**: This does not have to be equally distributed in every meeting, but it does have to even out over a period of time.
- **Social empathy that creates mutual understanding**: There is a high level of social sensitivity; team members can empathize with others and are attentive to their needs.
- **Mistakes are seen as a learning experience**: This important aspect has already been discussed in detail in the chapter on Agile Approaches in Section 16.3.7. It is important to learn from mistakes and weaknesses, and to seek solutions rather than culprits.
- **Curiosity and experimentation are encouraged**: Individual strengths, talents, and abilities are valued. In a team, everyone has their own talents and experience. The focus moves away from a deficit-oriented to a strength-oriented view.

It is important to emphasize that psychological safety does not mean that there should be no controversial or critical discussions. It does not mean being always nice to each other. However, the aim is always to treat each other with respect and appreciation.

Creating a work environment that provides psychological safety for everyone is not a no-brainer and requires effort. It starts, for example, with appreciative everyday communication between team members and also leaders. In particular, this includes giving and accepting feedback. Establishing an open and fruitful feedback culture can work wonders; a study by German personnel service provider Amadeus FiRe AG in co-operation with the Friedrich-Alexander University Erlangen-Nuremberg [75] has shown that the more satisfied employees are with the feedback culture in their organization, the more satisfied they are with their work in general. Psychological safety is also increased when leaders show interest in their team members, offer them opportunities for exchange beyond work-related topics, and convey that mistakes can also happen to themselves and mistakes are simply human. This means that they frame the work as a learning problem, not an execution problem.

23.5 Intercultural Collaboration Skills

Today's organization are often multi-cultural networks. While large organizations are spread across different continents and thus automatically have local sites near the roots of different cultures, small organizations will still need to do a worldwide search for experts to hire or suppliers to subcontract. We saw in Chapter 12 that system architects communicate and collaborate with multiple stakeholders in the organization's network. Today, this often involves meeting different cultures. Therefore, intercultural collaboration skills are needed. In this section, we will therefore briefly discuss intercultural collaboration. The bad news is: even Hampden-Turner and Trompenaars, leading researchers in investigating multi-cultural collaboration, have to admit about today's two major research points of view that they are "inadequate" [98]. They encourage their readers to think for themselves. So this is what we do as readers of Hampden-Turner's and Trompenaar's publications. Based on our own observations, conclusions, and experience, we present our own pieces of advice for those entering multi-cultural dialog:

- **Expect "culture" to have multiple dimensions**: So far we have talked about multi-national networks, which are the most trivial example of a multi-cultural environment. However, "culture" is not only something we can attach to the provenience of individuals in a geographical or national sense. We can also see different corporate cultures, different cultures in different political parties, different cultures in different religions, and even different cultures in software engineering education as opposed to mechanical engineering education. All these kinds of cultural differences can lead to misunderstandings that in our

opinion often have their root cause in missing awareness about the difference. It is this awareness that can help overcome misunderstandings, as we see in the next paragraph. A very rough advice can be: In the first place do not expect anyone to have the same cultural background as you have, even not people who share your mother tongue.

- **Identify common goals**: Once I am aware that others have a different approach, I can maybe see that in some areas I am striving for the same goal as they are, just in a very different fashion. Once I am able to identify common goals with my dialogue partners, it becomes much easier to negotiate those topics around which the alignment of goals is still pending.
- **Expect others to be different**: We have seen many situations in which people from culture A met people from culture B and just went into normal business as they would do it with a member of the family that has been their family's neighbor for the last three generations in the village in which they grew up. In some of these cases, we later heard about difficulties in reaching consensus. If you have reason to believe that people with whom you enter dialogue have a different cultural background than you, better be prepared for the extra need for clarifications and be extra cautious to stay open-minded and flexible in the way the dialog is conducted. If you do not understand what "they" say, just ask what they mean.
- **Avoid using metaphoric language in multi-cultural dialogue**: As discussed earlier in Section 23.1.3.4, different people can associate different meanings to the same word. This is even more relevant for metaphors, because they often make an implicit reference to the literature or the history of one culture. For example, the phrase "to meet one's Waterloo" means to meet a major defeat. It makes implicit reference to the final defeat of a person called Napoleon that was the result of a battle around a place called "Waterloo" [142], which today belongs to Belgium. The British named "Waterloo Station" according to this event in which they were on the winner side. Many British can probably relate to the phrase. Napoleon was French. Since none of the authors has this nationality, we do not even dare to predict how a French man or woman would feel about the words "to meet one's Waterloo" – and in most of the world, these words would probably have no meaning at all. We recommend to avoid any metaphor in multi-cultural dialogue. Since metaphors are supposed to explain something, it must be possible to explain the same thought in simple words and explicit technical terms instead of obfuscating it with a metaphor. Instead of talking of "meeting one's Waterloo," we can talk of "failing."
- **Be careful with humor or irony**: Trompenaars and Hampden-Turner [248] discourage the use of irony, because it is based on saying the opposite of what is meant. They also report different perceptions of humor by business people

from different cultures. Even though humor can be a good facilitator in a well-established team with a mutual understanding of the limits of what is funny, we thus recommend to be very restrictive on the use of humor in new intercultural setups.

- **If you think "they are stupid" then you may be trapped in ignorance**: After meetings between people from different cultures, we may interpret from some participants' evaluation that they considered the other participants to be unqualified or not applying the right mindset to the task at hand. Often it indeed occurs that approaches between different meeting participants are *different*, but this does not automatically imply that one approach is less appropriate than another one. On the contrary: being able to find different approaches is a strength, because it widens the solution space. We have seen differences in the perception of what is the most intuitive approach come up when cultures collide, and we recommend to everyone who thinks "the others' approach is stupid" to rethink into "their approach is different." This is a first step for getting into effective collaboration with those proposing the different approach. If you find yourself thinking that others propose a stupid approach then ask them why they suggest the approach rather than to finalize your conclusion that you will not pursue it.
- **Read Kissinger**: We already referenced the book "Diplomacy" by Henry Kissinger [142] when talking about Napoleon's final defeat. Even though it about multi-cultural phenomena in world history and politics, it gives some insights on typical multi-cultural misunderstandings that should be avoided. You may perceive that Kissinger expresses his very own point-of-view in his book. In case this happens and you disagree with that point-of-view then this trains you to accept the existence of different points-of-view, which is a non-negotiable prerequisite to enter any dialogue, especially the multi-cultural one.

Last but not least, keep remembering that we need multiple cultures and diversity to succeed with our business. Probably inter-cultural dialog will still stay one of the challenges in future and will not always be easy, but as long as we keep assuming that multi-cultural work produces more competitive results than mono-culture, we can hopefully motivate ourselves to enter the difficult but necessary dialogue with other cultures.

24

Outlook: The World After Artificial Intelligence

In this book, the fictitious "Scalable Observation and Rescue System" allows for studying today's systems architecting challenges with artificial intelligence inside the system-of-interest. Artificial intelligence however is expected to break out of the boundaries of such systems *under* development, also conquering the systems used *for* system development, e.g. the tools used by engineers and thus also the ones assisting the system architect. The superiority of machine execution of algorithms over manual architecting is seen already today when we watch functional architecture being created almost automatically during the semi-automatic process we have described in Figure 17.15. The only manual step remains the functional grouping – and who knows, maybe this step will soon be enabled by artificial intelligence and thus be machine-executable as well.

The frontiers between humans and machines move – and we may ask ourselves which of our own work in systems architecting will soon be done by an artificial intelligence. When talking about "The World After Artificial Intelligence" in the chapter title, we do of course not expect that artificial intelligence will ever vanish. It is simply too powerful. With the "world after," we mean the world in which artificial intelligence is an integral part of our daily life like electric fridges today – so a standard technology rather than still something under radical and game-changing development.

How will the world "after" artificial intelligence look like, a world in which humans even don't think about artificial intelligence, because it is so common as cold milk from the fridge today? Will there still be system architects in this future world, and what will they do?

We do not even dare to predict what human system architects will do in this future we imagine, but we dare to postulate that they will be there. This is because they have something that a machine will never have at its full disposition: the "tools" we described in Chapter 15. Remember that these tools were based on human reason enabling navigation inbetween politics and technology – and on leadership to make others follow the voice of reason. They were about facilitation

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

to align stakeholders. One may argue that machines can be taught reason and that the architecture stakeholders will be just other artificial intelligences in the future. But will we really want essential systems in our world to follow the reasoning of machines? The authors do not believe so.

This is why we believe that systems architecting will also in the future be driven by humans having the role we call today the system architect. When you enter the next debate about tools to be used in modeling, in tracking work items, in reporting your results – maybe relax, sit back, and think about the real tools that will make you the successful system architect also in the future: reasonable systems thinking and your soft skills.

We expect that systems architecting work whatever it will be like is going to be one of the key factors for selling better products successfully – hopefully based on human reason and on model-based systems architecting.

Appendix A

OMG Systems Modeling Language

The Object Management Group Systems Modeling Language (OMG SysML) [189] is a modeling language for model-based systems engineering. It supports and enables the specification, analysis, architecture and design, verification, and validation of a system. SysML defines the notation, semantic, and abstract syntax (data structure) of model elements and a set of diagram types. Diagram types are parts of viewpoints to create views on the model (see Section 8.2.1). The diagrams are clustered in structure and behavior diagrams and the requirements diagram (Figure A.1).

SysML is based on the Unified Modeling Language (UML) [188]. Both languages are defined and maintained by the Object Management Group (OMG). In 2001, the International Council on Systems Engineering (INCOSE) decided to establish UML as a standard modeling language for systems engineering. At that time, no standard modeling language for systems engineering was available, and UML was already widely spread and used in software engineering and partly in systems engineering. Tools, educated engineers, and best practices for UML were available. To avoid overloading the language, it was decided not to add a systems engineering perspective to the UML. Instead, a new modeling language using the profile extension mechanism of the UML should be developed. As a result, SysML 1.0 was published in 2007 as an OMG specification [183]. Formally, SysML is a profile of UML, although it is treated like a modeling language on its own.

SysML adds new model elements that are missing in UML like requirements or that are specific for systems engineering like blocks. SysML also removes elements from the UML vocabulary that are not useful in systems engineering, like classes or components that are specific for software engineering.

In summary, SysML is much smaller than UML. For example, SysML has nine and UML 14 different diagram types. In the following, we give a brief description of

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

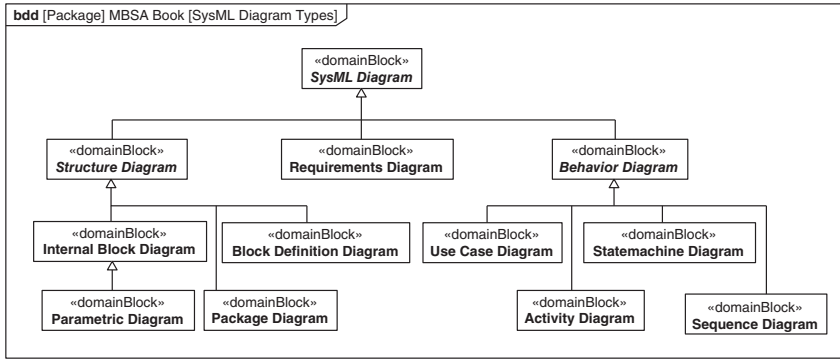


Figure A.1 SysML diagram types.

each SysML diagram type. A detailed description of SysML, including a modeling methodology can be found in [267, 269]. This chapter gives only a brief overview and cannot replace a complete book about SysML.

In 2017, the OMG released the request for proposals (RFPs) for SysML v2, kicking off the development of the next-generation modeling language. Since then, a group of experts has been working on the specification of SysML v2. At the time of writing this book, SysML v2 had not yet been published. However, Section A.7 provides a brief outlook on the new modeling language.

A.1 Architecture of the Language

SysML reuses a subset of UML and adds some model elements as stereotypes. In summary, SysML is a smaller modeling language than UML (Figure A.2).

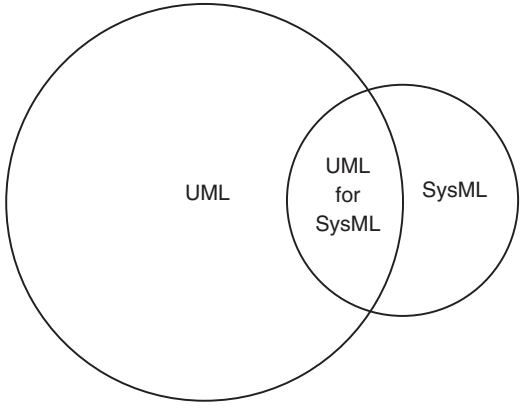


Figure A.2 UML for SysML.

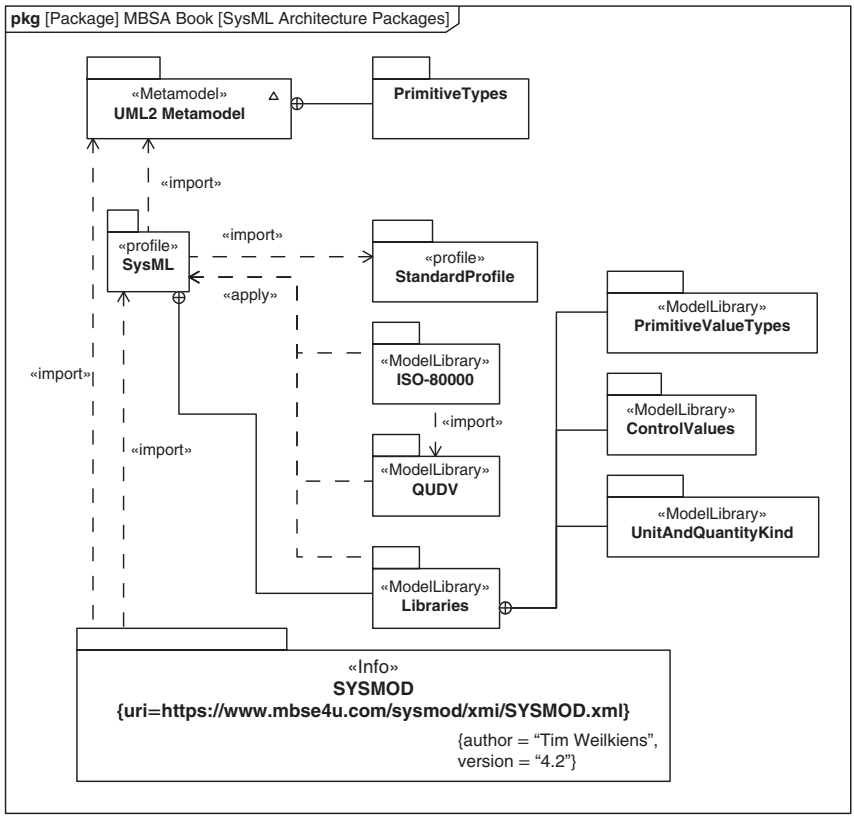


Figure A.3 SYSMOD, SysML, and UML.

Figure A.3 shows the relationship of SysML to related packages of the modeling landscape. The SysML profile imports the UML metamodel and the standard profile. The standard profile is a set of stereotypes that are specified in the UML specification and are commonly used. The SysML stereotypes extend UML metamodel elements or specializes stereotypes from the standard profile. For example, the SysML trace relationship is a specialization of the trace stereotype from the standard profile (Figure A.32 on page 375).

SysML also defines some model libraries. The "Model Library for Quantities, Units, Dimensions, and Values" (QUDV) provides the elements mentioned in the title of the library. The "ISO 80000" library provides units and quantity kinds from the corresponding ISO standard.

The library "PrimitiveValueTypes" contains the values types "Real," "Integer," "Complex," "String," and "Boolean." The library "ControlValues" contains special

elements for activity modeling, and the model library "UnitAndQuantityKind" defines the types to define units and quantity kinds.

A.2 Diagram and Model

SysML follows the principle of separation between the model and the representation of the model, also called "View and Model" (Section 9.7). The language defines a semantic and abstract syntax (model) and a notation or concrete syntax as well as a set of diagram types (viewpoints). The information of a SysML diagram, like the position or size of an element, is stored separately from the model information. The left side in Figure A.4 depicts the abstract syntax. For example, the use case "Book a Tour." The concrete syntax, which means the ellipse notation of the use case, is not shown. The right side in Figure A.4 shows parts of the appropriate diagram syntax, which is the standardized data structure of the layout information.

Views show only the information that is intended by the model builder. They are, typically, incomplete according to the information that is stored in the model.

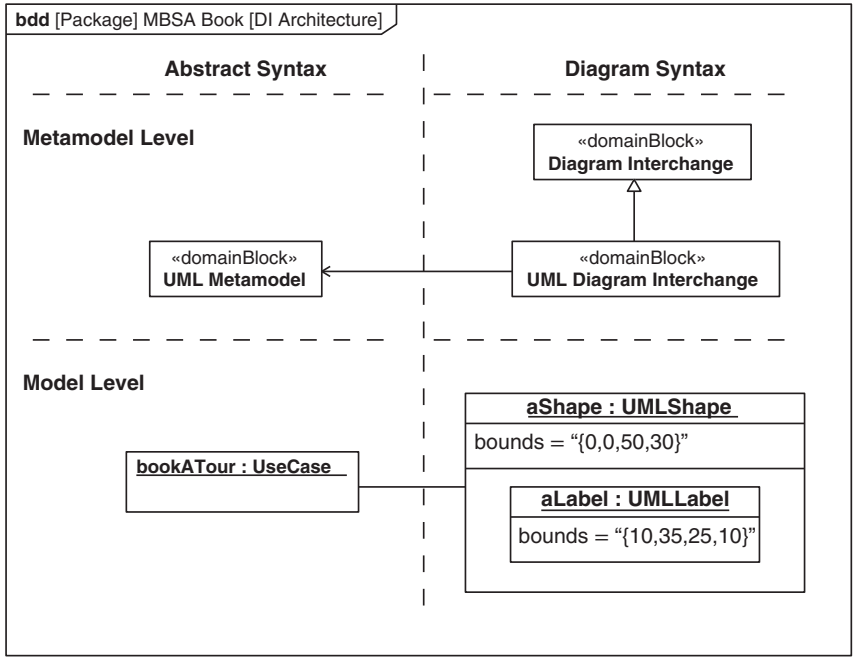


Figure A.4 Diagram interchange.

The separation of the view and model is obvious, for example, in an application like Excel, where the data in the sheets are the model, and the diagrams are the views on the data. The Excel author edits the information directly in the model, and the diagrams are exclusively used for presentation purposes.

It is different for SysML. Typically, the model builder creates and edits information using a diagram and not the model itself. The model gets into the background, and the primary artifacts are the diagram. Therefore, it is important to emphasize that a SysML diagram is only a view and the real information lies in the model.

The nine SysML diagram types are standard viewpoints. However, it is not forbidden and explicitly desired that you create additional viewpoints, for example, PDF documents or spreadsheets (see also Chapter 8).

The header of a SysML diagram follows the syntax `<diagram type> [<model element>] <model element name> [<diagram name>]`. In Figure A.4, the diagram type is "bdd" (= block definition diagram), the model element is a package, the name of the package is "MBSA Book," and the name of the diagram is "DI Architecture."

A.3 Structure Diagrams

The structure diagrams are a set of diagrams to provide a view on the structure of the system (Figure A.5). The main diagrams are block diagrams, the block definition, and the internal block diagrams.

A package diagram shows the namespace structure of the model. A parametric diagram is a special internal block diagram and shows the parametric relationships between value properties of the system.

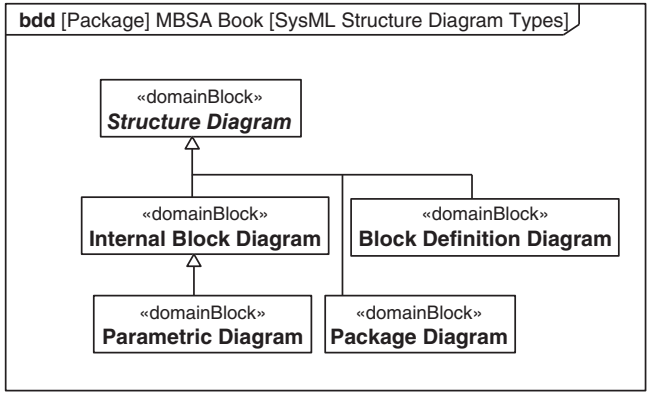


Figure A.5 SysML structure diagram types.

A.3.1 Block Definition Diagram

As stated in the name, a block definition diagram depicts the definition of blocks, which means the blueprints of the physical or virtual system parts. A block defines the structure and behavior of an entity and can have value properties, operations, constraints, part or shared properties defined by other blocks, and an internal structure.

Figure A.6 shows the part properties (parts) in a compartment of the owning block. Figure A.7 shows the parts of the "Tour Robot" with the composite relationship notation which is a directed association with an arrow at the end of the defining block of the part and a black diamond symbol at the end of the owner of the part. Figure A.6 depicts the block "Tour Robot" with value properties "range" and "mass" including units, a constraint for the weight, and an operation "getBatteryCapacity()" that returns the battery capacity of the tour robot in Ah.

Besides the association, the generalization is another common relationship in a block definition diagram. It is a solid line with a hollow triangle as an arrowhead. You can see a generalization relationship, for example, in Figure A.7 in the upper left corner between the "Virtual Tour Server" and the "VT Server." The virtual tour server is an element of the logical architecture and has a generalization relationship to the abstract VT server from the base architecture. A generalization in one direction is a specialization in the other direction. It depends on your viewpoint.

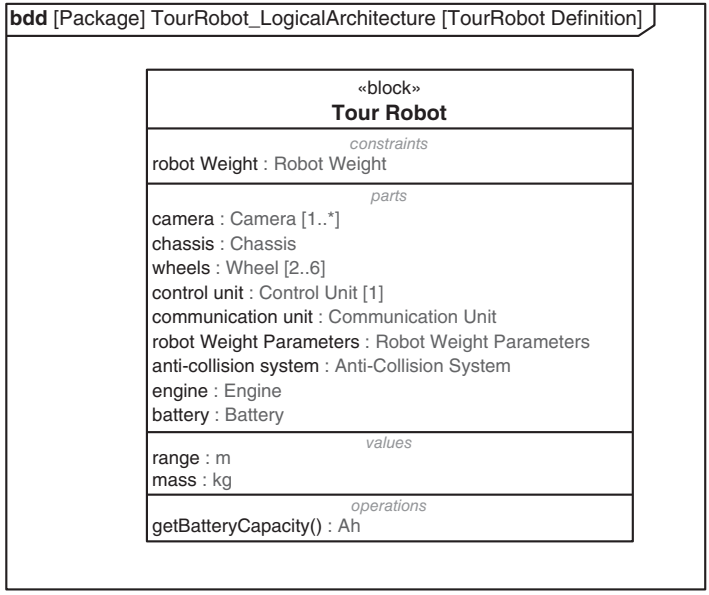


Figure A.6 SysML block.

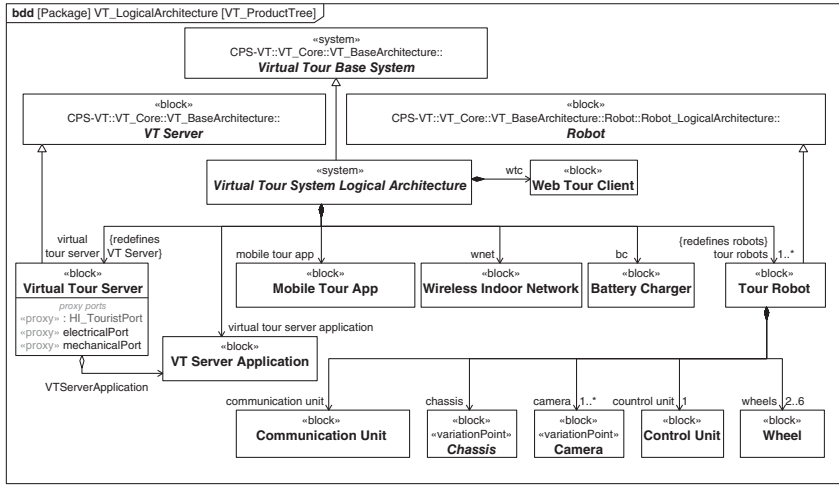


Figure A.7 Example Product Tree of the Virtual Tour system.

An abstract block means that the definition of the block is generic, and it needs additional information for a concrete definition of a real-world entity. The name of an abstract block is set in italics. The virtual tour server inherits all features from the abstract VT server due to the generalization relationship. Frankly spoken, all features are copied from the general block to the special block. It is possible to redefine inherited features. In summary, only both blocks together define the complete virtual tour server entity.

Two common applications of the block definition diagram are the product tree and the domain knowledge model. The product tree is a tree-structured breakdown of the system (Figure A.7). The diagram focusses on the list and hierarchy of the system blocks. Therefore, the details depicted in compartments of each block are, typically, hidden.

A domain model defines the domain knowledge of the system (Section 10.2.5). Figure A.8 shows an extract of the domain model. A more complete diagram is shown in Figure 10.11 on page 111. The blocks of the domain model have the stereotype «domainBlock» from the SYSMOD profile [267, 269] to mark them as a special kind of a block. The solid line between the blocks is an association without composite semantic that defines properties at both ends. The respective property relates to the block on the other side of the association. Associations are read in the following pattern: "At any time, every <block> has <multiplicity> <block> in the role <role name>." This sentence must be meaningful for the domain. Let's test it for our example in Figure A.8: "At any time, every customer has any number of tour bookings in the role booked tours." Though this sounds a bit clumsy, it makes sense.

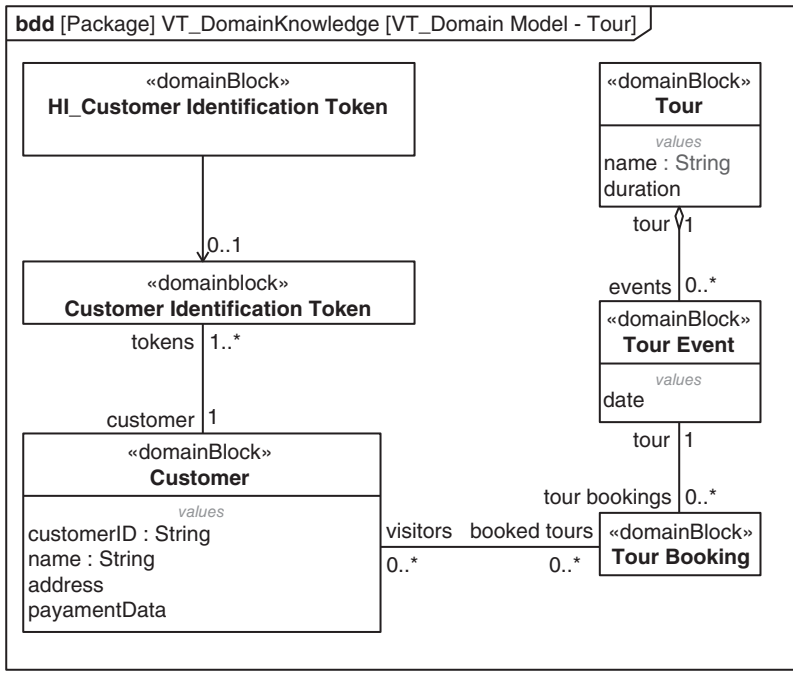


Figure A.8 Example domain model of the virtual tour system.

The association between the block "Customer" and the block "Tour Booking" in Figure A.8 defines a property named "visitors" of type "Customer" with multiplicity [0..*] and a property named "booked tours" of type "Tour Booking" with multiplicity [0..*]. The property visitor is a reference property of the block "Tour Booking," and the property "booked tours" is a reference property of the block "Customer." Unlike the composite part property described above the reference property is not owned by the block. The association between "Tour" and "Tour Event" has a hollow diamond at one end which indicates a shared aggregation. It is like an association with the added semantics that the end of the hollow diamond symbol is an aggregation of shared things as opposed to the composition, which is an aggregation of owned things.

A special application of the block definition diagram is the activity tree. An activity (Section A.4.2) without its internal structure (actions, flows, and control nodes) can be shown like a block in a block definition diagram. It is depicted as a rectangle with the name of the activity and the keyword «activity». Activities are modeled in a tree structure with a composition relationships (Figure A.9). The tree depicts a call hierarchy and not an ownership hierarchy like the product tree.

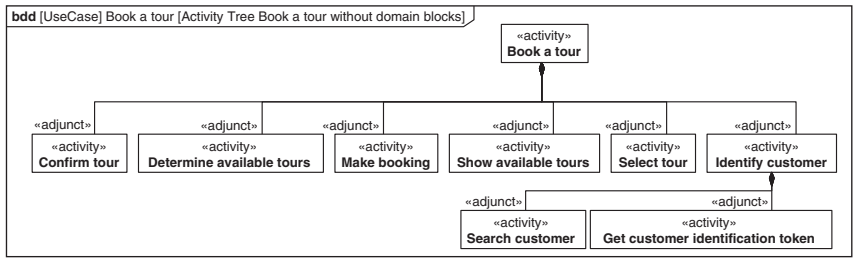


Figure A.9 Activity tree.

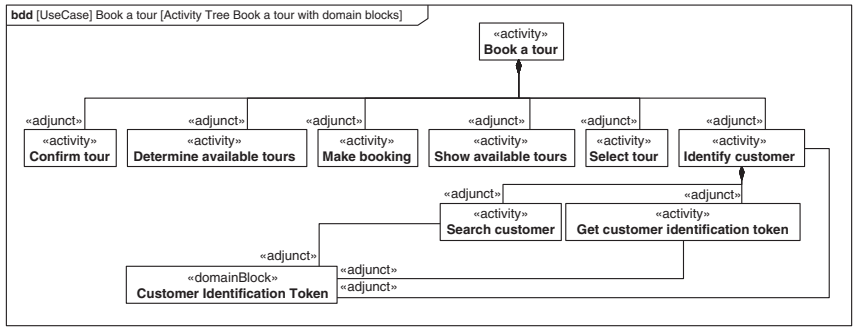


Figure A.10 Activity tree with associated blocks.

The semantics of composite relationships between activities specifies that the activity at the end of the black diamond calls the activity on the other end by a call behavior action. The calling activity owns the execution instance of the called activity. Properties at the end of the called activity are stereotyped by SysML «adjunct». Adjunct properties constrain the values of properties determined by the calling activity. Input or output parameters of activities can also be represented with association relationships, as shown in the example in Figure A.10 using the "Customer Identification Token."

A.3.2 Internal Block Diagram

While block definition diagrams depict the definitions of blocks, internal block diagrams show the internal structure of a block, that means the properties, their connections, and the used interfaces between them.

Figure A.11 shows a part of the logical architecture of the Virtual Tour system in an internal block diagram. The rectangles in the diagram are properties of the corresponding block. Here, the enclosing block is the "Logical Virtual Museum Tour System," that is, the root node of the logical architecture, as you can see in

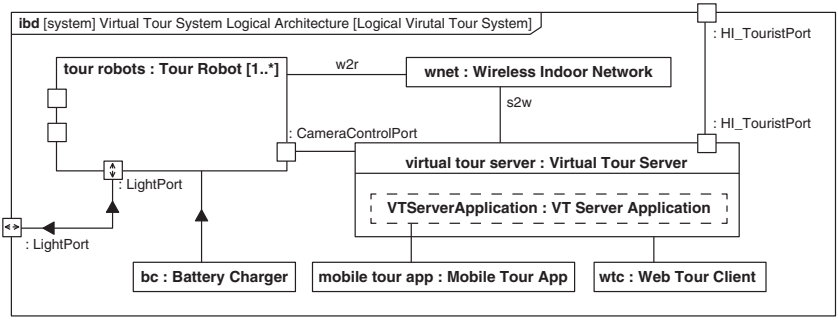

ibd [system] Virtual Tour System Logical Architecture [Logical Virutal Tour System]
: HI_TouristPort
tour robots : Tour Robot [1..*]
w2r
wnet : Wireless Indoor Network
s2w
: HI_TouristPort
: CameraControlPort
virtual tour server : Virtual Tour Server
: LightPort
VTServerApplication : VT Server Application
: LightPort
bc : Battery Charger
mobile tour app : Mobile Tour App
wtc : Web Tour Client

Figure A.11 Example internal block diagram of the virtual tour system.

the header of the diagram. The syntax of the header is explained in Section A.2. The diagram frame represents the border of the enclosing block.

Properties of blocks are also called parts when they are typed by a block and defined as a composite property. For example, "tour robots" is a part of the "Logical Virtual Tour System." A part has a name (for example, "tour robots"), a type (for example, "Tour Robot"), and a multiplicity (for example, [1..*]). The textual syntax is "tour robots:Tour Robots[1..*]." The definitions of the properties are shown in the block definition diagram in Figure A.7 on page 355. Non-composite properties, so-called reference properties, are shown with a dashed rectangle.

The solid line between properties is a connector. It represents that there is some kind of interaction or exchange of items between the connected properties. Connectors can have a name. For example, "w2r" and "s2w" in Figure A.11.

The exchanged items can be explicitly specified by the item flow. The item flow is depicted by a black triangle to indicate the direction and a text nearby that describes the item. An item is a property of the surrounding block or simply the type that flows. In Figure A.11, the item flow specifies that power flows from a battery charger to the tour robots.

The small rectangles on the border of the property rectangles are ports. They are special properties and are defined at the corresponding block of the property and represent an interaction point of the block to its environment. Like a property, a port has a name and a block as its type. The block defines the properties that can flow through the port and defines the provided and requested properties and operations. Flow properties are specified with a direction "in," "out," and "inout," and features like operations are specified if they are provided "prov" or requested "reqd" (Figure A.12).

SysML differentiates between full ports and proxy ports. A full port is similar to a part that is placed on the border of the enclosing block. It represents a real element of the bill of material (BOM) that is an interface of the enclosing block. A full port is shown with the keyword «full».

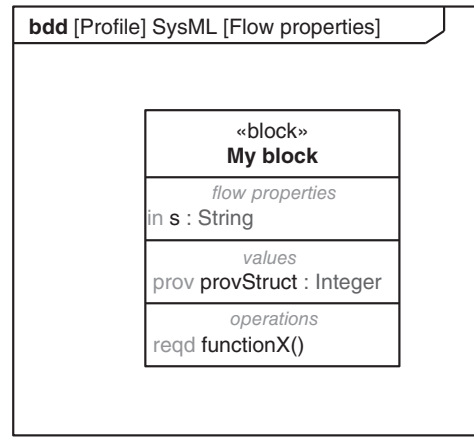


Figure A.12 Flow properties and provided/requested features.

The proxy port is a placeholder for internal parts or ports at the border of the enclosing block. Since it is only a placeholder, it is not an element of the BOM.

A connector between a proxy port and an internal part is a binding connector. It defines that the connected ends are identical, which means that the placeholder proxy port always represents the real element.

We recommend to only use proxy ports and no full ports. You can replace every full port with a proxy port and the appropriate internal part, as shown in Figure A.13. But you, typically, will always need proxy ports in your model to show deep nested interfaces at the border of a block. Figure A.11 shows proxy ports at the border of our system block "Logical Virtual Tour System" that represents interfaces of inner parts. The system element itself has no real interfaces. If you do not use any full ports in your model, you can hide the keyword «proxy» at the proxy port in the diagram.

Instead of being connected to internal properties, a proxy port can also be a so-called behavioral port. That means that any requests arriving at the port are handled by the block that owns the port. Ports of the functional parts in Figure A.14

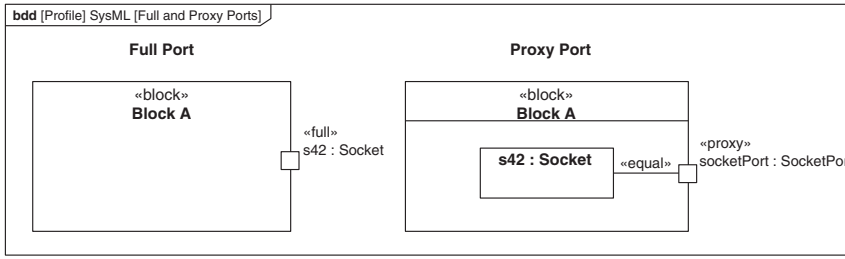


Figure A.13 Full and proxy port.

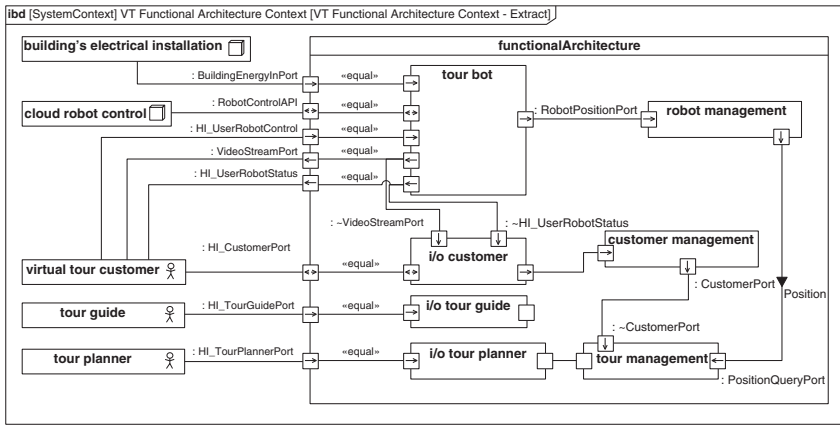


Figure A.14 Behavioral proxy ports in the functional architecture.

are behavioral proxy ports. A behavioral port could have an additional small state symbol inside the enclosing block attached with a solid line to the port. We do not use that notation in Figure A.14.

The port typed by VideoStreamPort at the part "i/o customer" in Figure A.14 is a conjugated port as depicted by the tilde symbol (~) as a suffix to the port type. A conjugated port reverses the directions of the features and flow properties of the port type, which means "in" turns to "out," "prov" to "reqd," and so on.

Figure A.15 shows some types of the proxy ports. They are interface blocks that specify provided and required operations and properties as well as flow properties. They are only specifications of interaction points and do not represent real entities. User interfaces are stereotyped with «userInterface», a special interface block of the SYSMOD profile [267, 269] specifying human-machine interactions.

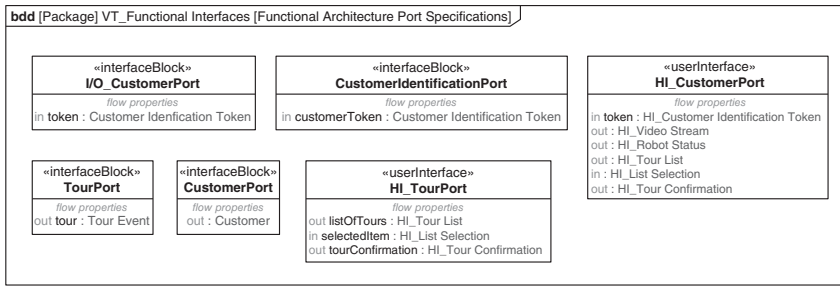


Figure A.15 Interface block specialization.

A.3.3 Parametric Diagram

A parametric diagram depicts relationships between property values of the system. The related properties do not need to be defined in the same block. However, the parametric diagram is a special internal block diagram, and the appropriate enclosing block must have access to the related properties. Typically, it is the first common node of the properties in the product tree or a specifically created context block that represents the context of the constraints.

In Figure A.16, the enclosing block is the Virtual Tour system context element. It is the lowest node in the tree that has access to properties of the tour robot and to properties of the exhibition that is a system actor of our system.

Constraint blocks define constraints and their parameters. Figure A.16 shows the constraint block "Range of Tour Robot" that refines the textual requirement "Range." The constraint block indicates the SysML stereotype «constraint» above its name. Parameters of the constraints are defined in a special compartment with headline "parameters."

Figure A.17 is a parametric diagram that depicts the constraint property of the constraint block "Range of Tour Robot." The squares attached at the inside of the constraint rectangle in a parametric diagram represent parameters of the constraint block. The solid line between a parameter and a property value is a binding connector that asserts that the values at both ends are equal. Undefined

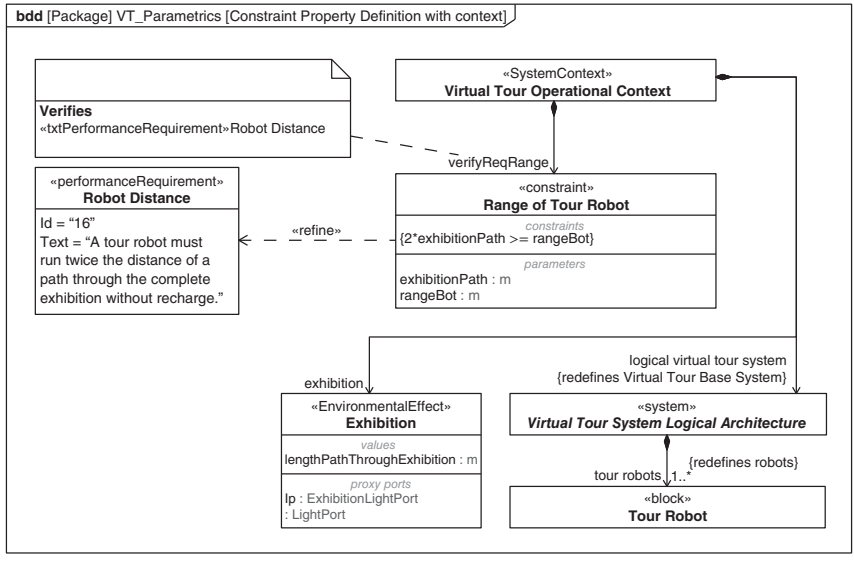


Figure A.16 Definition of a constraint property.

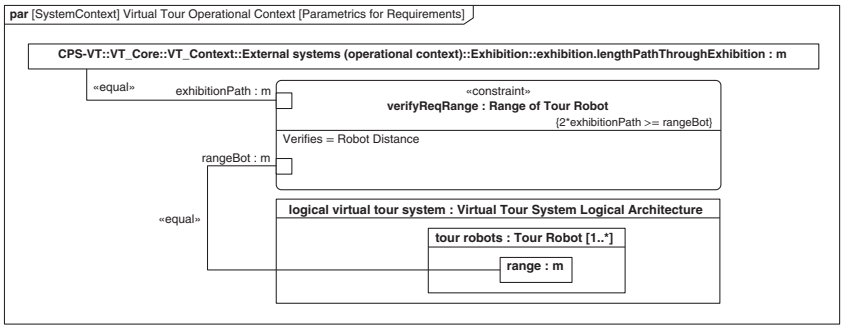

par [SystemContext] Virtual Tour Operational Context [Parametrics for Requirements]
CPS-VT::VT_Core::VT_Context::External systems (operational context)::Exhibition::exhibition.lengthPathThroughExhibition : m
«equal»
exhibitionPath : m
«constraint»
verifyReqRange : Range of Tour Robot
{2*exhibitionPath >= rangeBot}
Verifies = Robot Distance
rangeBot : m
«equal»
logical virtual tour system : Virtual Tour System Logical Architecture
tour robots : Tour Robot [1..*]
range : m

Figure A.17 Example of a parametric diagram.

values could be calculated by a modeling tool if the equations are solvable. For example, you can calculate the minimum range of a tour robot by adjusting the parameter for the length of the path through the exhibition.

The constraint property "verifyReqRange" in Figure A.17 describes how to verify a solution and becomes a test case. A SysML verify relationship connects a test case with a requirement and documents against which requirement the constraint verifies the solution. The verify relationship is shown with a dashed line with an open arrowhead and keyword «verify». Figure A.16 depicts the alternative call-out notation. It can show the information of the verify relationship if only one of the related elements is shown.

A.3.4 Package Diagram

Packages provide a generic model organization capability. Typically, a model consists of hundreds or thousands of elements. They must be structured to get a better overview, to work with several people on a single model, and to reuse parts in other models. Packages are like the directories on your hard disk to organize the files. Similar to file browsers, modeling tools have a model or project browser to show the package structure of the model (Figure A.18). That tree notation in a browser is not part of SysML. Instead, SysML provides the package diagram to show packages and their relationships.

Figure A.19 shows the top-level packages of our Virtual Tour system model and the application of profiles (Section A.6). The package symbol with the triangle represents the model. It is a SysML model element similar to the package element.

Nested packages could be shown inside the owning package (Figure A.19). If you show the nested relationship explicitly, you can show the packages in a tree-like notation (Figure A.20).

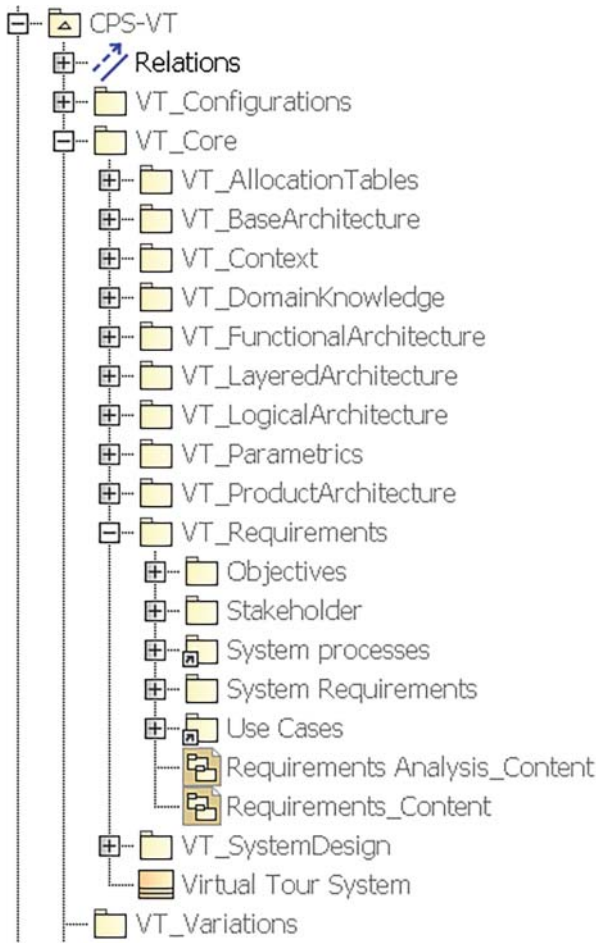


Figure A.18 Model browser of a modeling tool.

A.4 Behavior Diagrams

Behavior diagrams show behavioral aspects of the system. SysML defines four different behavior diagrams (Figure A.21). An activity diagram shows flow-oriented behavior, a state machine diagram event-oriented behavior, and a sequence diagram message-oriented behavior. A use case diagram is an exception. Strictly speaking, it is not a behavior diagram but a structure diagram. It does not show how behavior is specified but a list of top-level system behaviors. Anyway, SysML (and UML) still categories the use case diagram as a behavior diagram. It is a

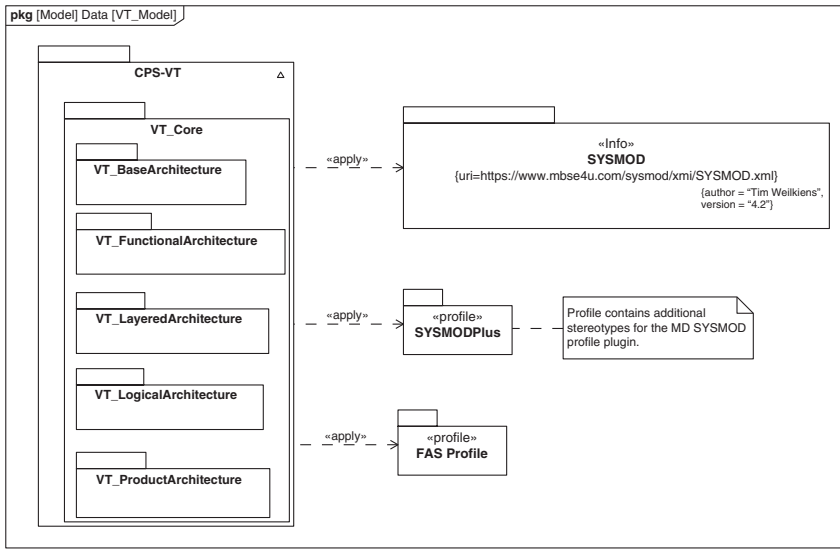


Figure A.19 Example package diagram of the VMT.

minor issue and not worth to fix it with regard to the effort and impact of the change in the specification and the modeling world.

A.4.1 Use Case Diagram

A use case specifies a set of actions that leads to an observable result that is of value for the actors or stakeholders of a subject. A use case diagram shows the use cases with their associated actors and relationships to other use cases. Typically, the specified behavior of a use case is described by an activity (Section 10.2.3).

Figure A.22 is a use case diagram that shows some use cases of the actor "Virtual Tour Customer" of our example Virtual Tour system. The sticky figure symbols represent actors. It is the standard symbol for actors in SysML. However, it is possible to use other symbols to visualize different actor categories. For example, a cube symbol is commonly used for nonhuman actors like an external system. You can define customized symbols by stereotypes. The stereotype «user» in Figure A.22 is part of the SYSMOD profile and not a standard SysML element.

Actors are associated with the use case by an association. It is the solid line that specifies that the actor is involved in the behavior that is described by the use case. It is the same relationship that is used between blocks (Section A.3.1).

«systemUseCase» and «continuousUseCase» are stereotypes of the SYSMOD profile to categorize use case types and are not part of SysML. See Section 10.2.3 for a description of how to use these special use case types.

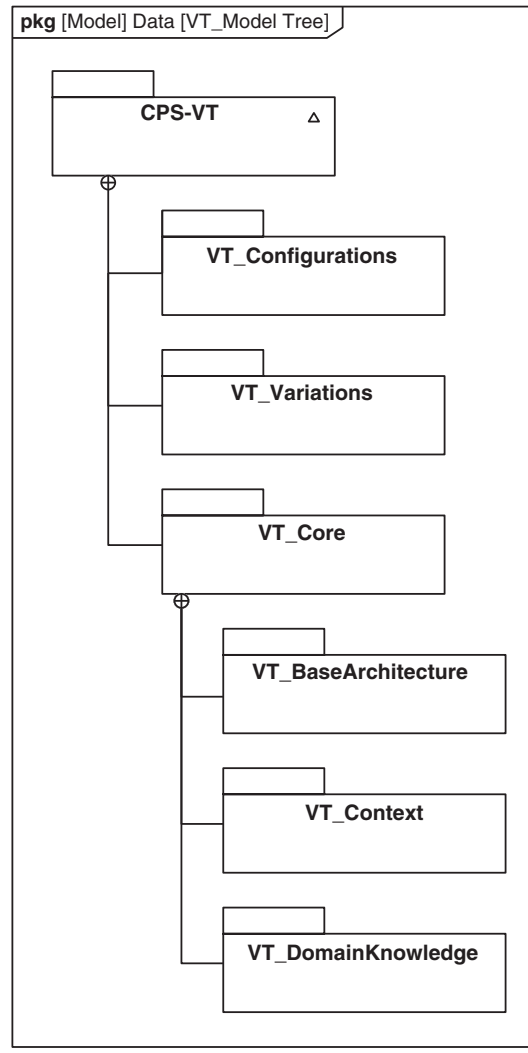


Figure A.20 Tree-like notation of nested packages.

The subject of a use case can be visualized in the diagram by a box around the use cases (Figure A.23). Typically, it is the system-of-interest itself. In that case, the box is typically omitted.

In our example system, the tour robots are specified as if they were a standalone system. In particular, this means that there are use cases that refer to the tour robots. They are included in use cases of the Virtual Tour system, which can be modeled by the include relationship. At the activity level, this relationship is reflected in a corresponding call behavior action. However, this is

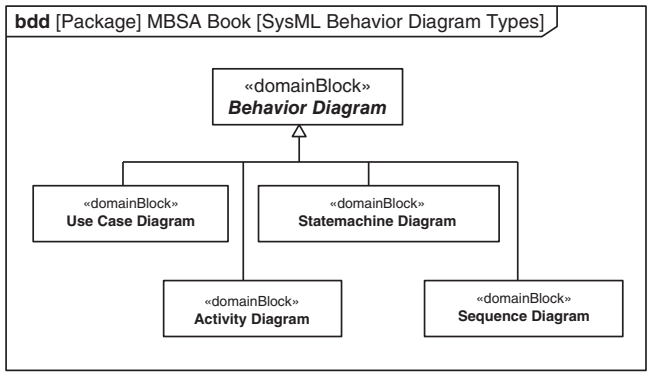


Figure A.21 SysML behavior diagram types.

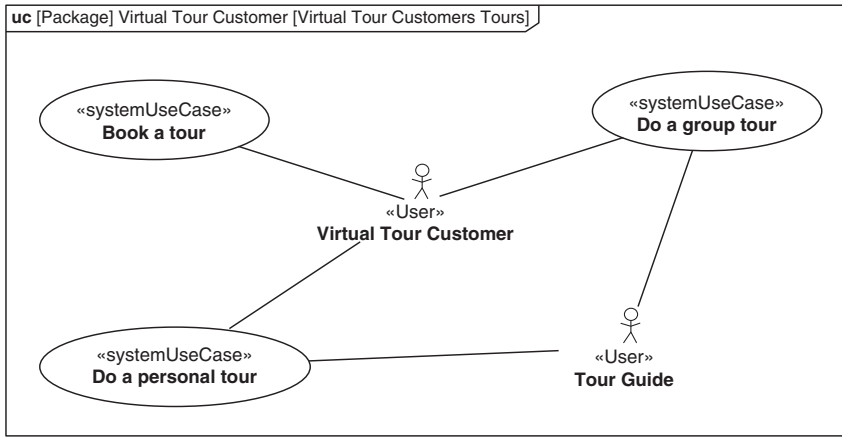


Figure A.22 Example use case diagram.

a methodological relationship and is not required by SysML and accordingly not implemented by the tools in an automated way.

A.4.2 Activity Diagram

An activity is a special behavior that specifies the execution order and the inputs and outputs of actions. An activity diagram depicts exactly one activity. The order of the actions is specified by control nodes, for example, fork or decision node, and by control flows depicted by dashed arrows. The relationship between inputs and outputs of the action is specified by object flows depicted by solid arrows.

A common application of activities is the specification of a use case behavior. Figure A.24 shows an activity diagram of the use case activity "Book a tour" of the

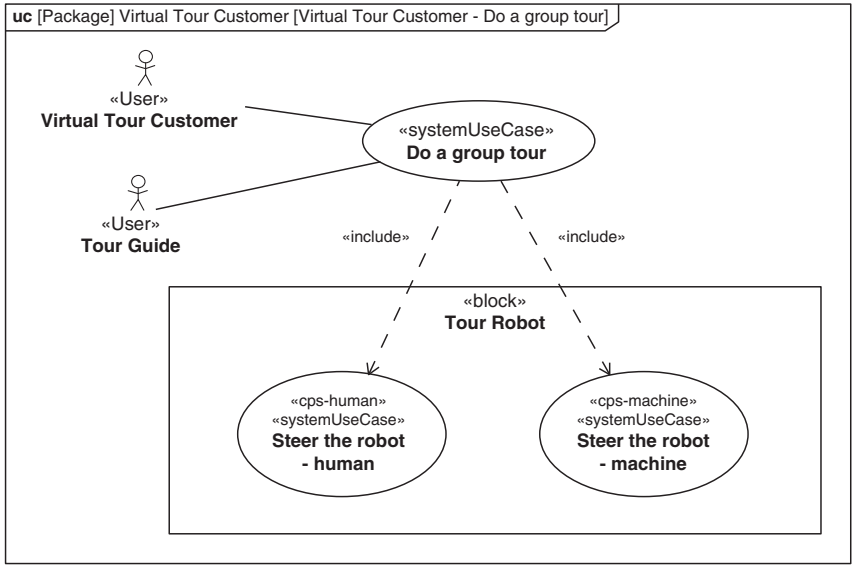


Figure A.23 Example use case subject and include relationship.

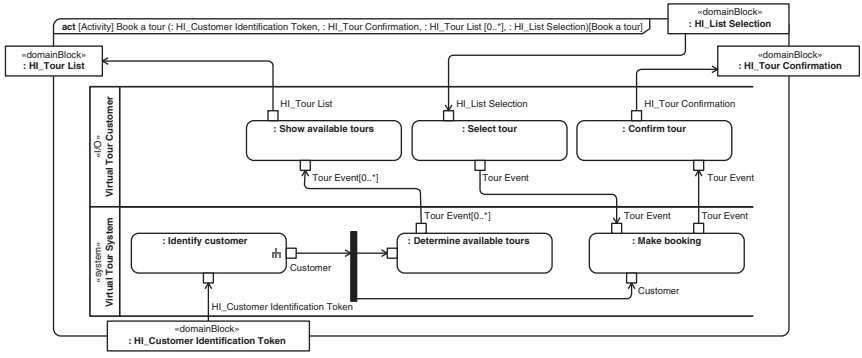


Figure A.24 Example activity diagram.

Virtual Tour system. The two horizontal lanes are activity partitions that group the included actions according to a given criterion. The model builder can define any kind of criteria. Here, it is the separation of input/output logic and core logic.

The rectangles at the border of the activity diagram represent the input and output parameters of the activity. The small rectangles at the border of the actions are input and output pins and represent the parameters of the action.

An action is an atomic piece of behavior. It is not further detailed in the model. SysML defines several action types in a formal way that makes it possible to provide

a common model execution environment for them and to run simulations. To close some gaps in the SysML, respectively, UML specification the OMG released the "Semantics Of A Foundational Subset For Executable UML Models" (fUML) [185] that, besides others, refines the action definition of UML.

If you do not execute your model, only a few action types are relevant. The opaque action specifies a behavior defined in any language. Besides formal languages like a programming language, it could also be a natural language like English. Figure A.25 shows an extract of an activity diagram with an opaque action "Identify customer." The specification language is English. It is possible to give the action a name in addition to the specification to avoid programming language code in the diagram and to make it easier for the model reader. In Figure A.25, the name and the specification are identical. Note that we recommend to use call behavior actions. See also Section 10.2.4.

The actions in Figure A.24 are called behavior actions. They call another behavior. Typically, it is another activity, but could also be a state machine or an opaque behavior. Call behavior actions have a small fork symbol in the lower right corner.

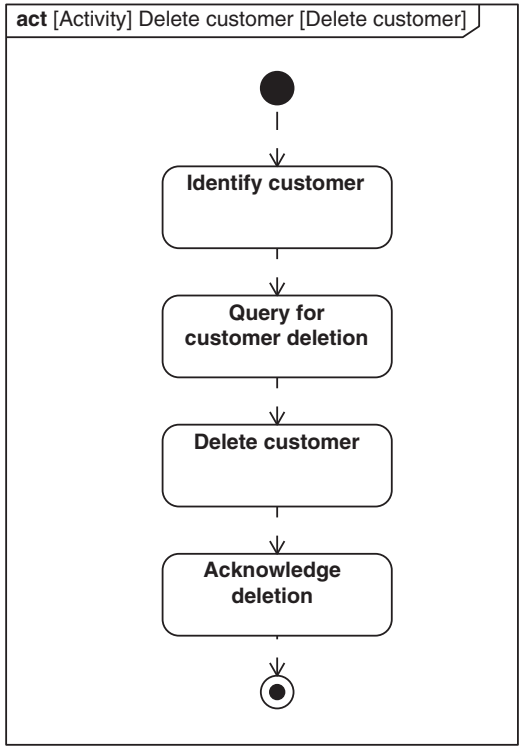


Figure A.25 Opaque actions.

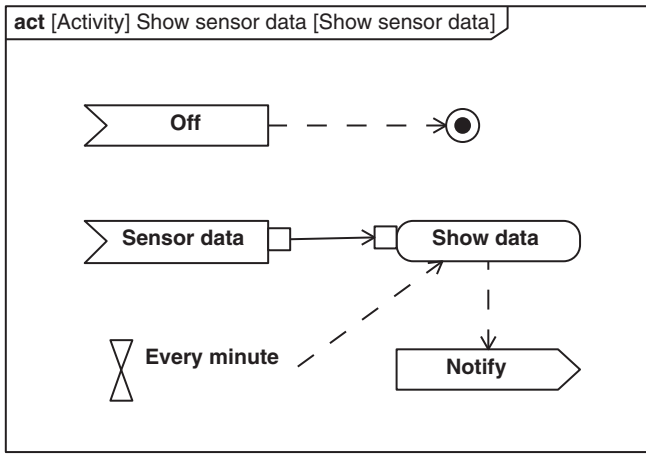


Figure A.26 Accept event action and send signal action.

Finally, you need actions to specify sending or receiving of signals. They are called accept event action and send signal action. Figure A.26 depicts an activity with three AcceptEventActions "Every minute," "Off," and "Sensor data," and a SendSignalAction "Notify." When started, the activity listens for sensor data and off signals and the timer signal "Every minute." AcceptEventActions that receive timer signals have a special notation like an hourglass. Each time the activity receives sensor data, the data are shown somehow when there is a token at the object flow from the timer signal available. Finally, the activity sends a notify signal. The recipient of the signal is not shown in the diagram but specified in the model. When the activity receives a signal "Off," the activity final node (black circle with outer circle) terminates the execution of the whole activity.

The thick black vertical line in Figure A.24 is a fork node. It is a special control node that splits one ingoing flow in two or more outgoing flows. The subsequent two actions are executed independently of each other and get both the same input object.

SysML provides much more elements for activity modeling. We have just provided the basic set to model most flow behavior of a system. For more information about activity modeling, see [267, 269].

A.4.3 State Machine Diagram

A state machine specifies discrete behavior through finite state transitions. It represents the states of an entity and the transitions between the states. A state machine diagram depicts a single state machine.

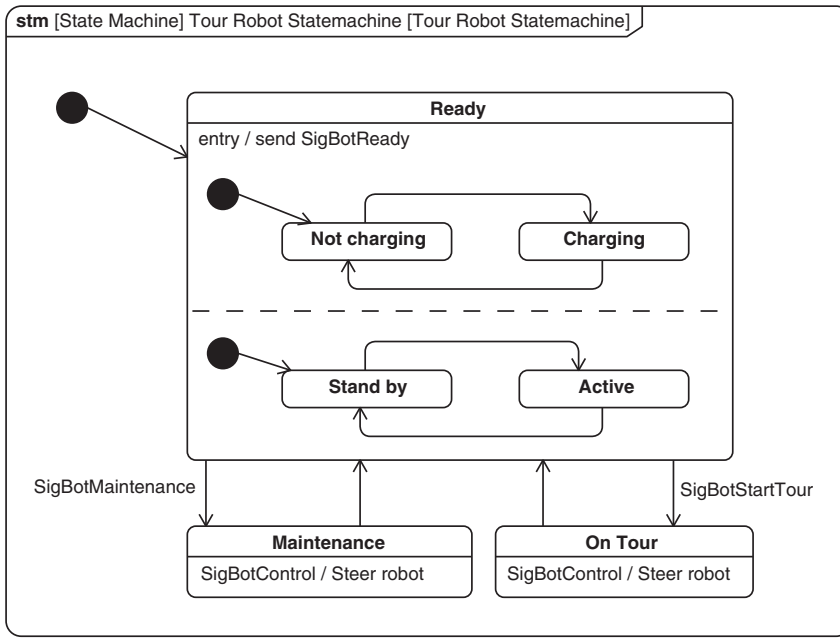


Figure A.27 Example state machine diagram.

Figure A.27 shows a state machine of the tour robot from our Virtual Tour system example. The black dot with the outgoing arrow is the initial state and points to the first state that will be active when the state machine is executed. The first state of a tour robot is "Ready." The state "Ready" has an entry behavior. The action is always performed immediately when the state is entered. It is also possible to define an exit behavior that is performed just before the state is left.

If the robot receives the "SigBotStartTour" signal, the appropriate outgoing transition "fires," and the robot is in the state "On Tour." Only in this state and in the "Maintenance" state, the robot reacts to the signal "SigBotControl" with the behavior "Steer robot." It is an internal transition, which means the corresponding state is not left during the transition. Each state could have substates to model details of the state behavior.

The state "Ready" has several substates in two orthogonal regions. The regions are separated by a dashed line. One region has exactly one active state during execution. For example, the tour robot can be in the states "Ready::Charging" and "Ready::Stand-by" at the same time. It is not allowed to model a transition from one region to another region. Besides this formal rule, it also makes conceptually no sense to have transitions between regions because they represent orthogonal concepts.

A.4.4 Sequence Diagram

A sequence diagram shows the exchange of messages between elements of the system. Typically, only a few scenarios are shown in a sequence diagram and not all possible paths like in an activity diagram (Section A.4.2). Although it is possible to model alternatives, loops, and parallel message exchanges. However, the diagram gets very confusing when you use those elements intensively. Areas of application of the sequence diagram are, for example, test scenarios, example scenarios, or detailed specifications of a communication protocol between elements.

Like an activity diagram shows an activity model element and a state machine diagram a state machine, a sequence diagram depicts an interaction model element.

Figure A.28 is an example of a sequence diagram of the Virtual Tour system. It shows a message exchange between the control server and a tour robot. The

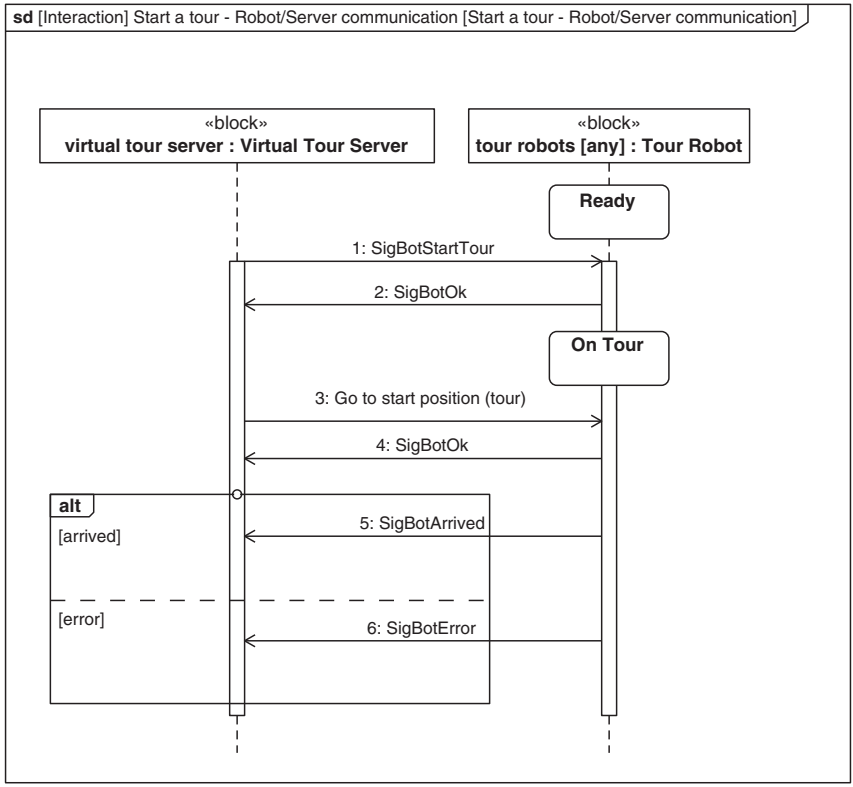


Figure A.28 Example sequence diagram.

"Virtual Tour Server" sends the signal "SigBotStartTour" to a robot. The robot responses with the signal "SigBotOk." Then, the tour server calls an operation at the robot to steer it to the start position. Again, the robot acknowledges the command with the signal "SigbotOk." Finally, the robot responds with the signal "SigBotArrived" when it has arrived at the requested position. In the case of an error, the robot sends a "SigBotError" signal with an error code. The rectangle around the two messages "SigBotArrived" and "SigBotError" is a combined fragment. It is used to model interaction operators like loop or break. Here, it is an alternative depicted by the keyword "alt" in the upper left corner of the combined fragment.

The rectangles with the vertical dashed lines are called lifelines. They represent one communication partner. It is an element from the usage level (Section 9.4), for example, a part property of a block. If the part property is multivalued, a selector expression defines which value is represented by the lifeline. In Figure A.28, the selector expression "any" selects a random tour robot.

The rectangles on the lifeline represent that the object represented by the appropriate lifeline is active. The arrows represent the messages. An asynchronous message has an open arrowhead like in Figure A.28, and a synchronous message has a filled arrowhead.

It is possible to show the respective state of the object by depicting the appropriate state symbols on the lifelines as shown in Figure A.28.

A.5 Requirements Diagram

A requirements diagram is neither a structure diagram nor a behavior diagram. It belongs to a diagram category of its own (Figure A.1).

A requirement in SysML is specified by a text. Although you use a graphical modeling language, a requirement's textual specification is not fully turned into lines and boxes by SysML. The requirement text is encapsulated in the SysML model element "Requirement." In addition, the model element stores the name and the unique identifier of the requirement. The applied requirement methodology could add specific requirement attributes like priority or maturity. They are not part of the standard SysML and could be added by using the stereotype mechanism (Section A.6).

There are many relationships available to connect a requirement with other requirements or other model elements. For example, a block can satisfy a requirement, a test case can verify a requirement, a requirement can be decomposed to other requirements, refined by a use case, or have trace relationships to other requirements.

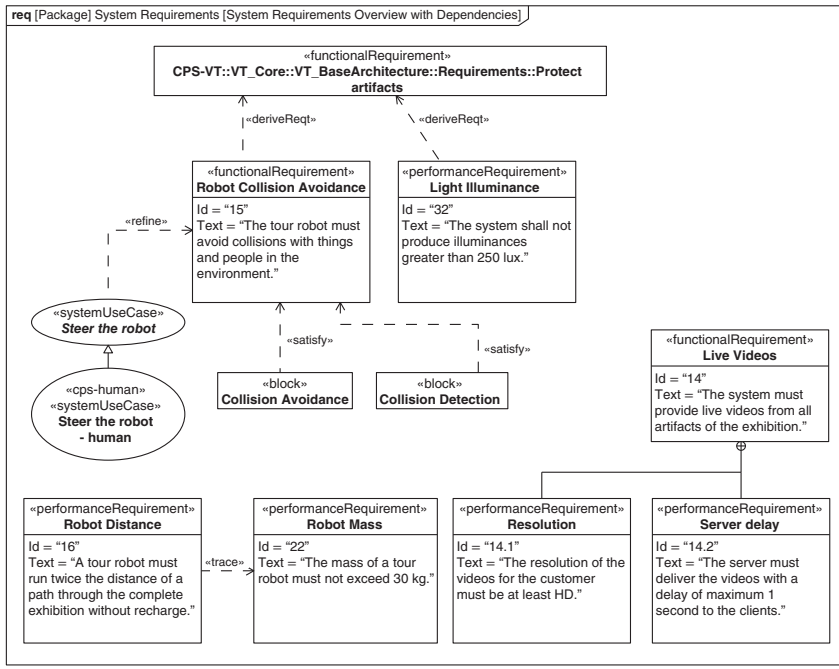

req [Package] System Requirements [System Requirements Overview with Dependencies]
«functionalRequirement»
CPS-VT::VT_Core::VT_BaseArchitecture::Requirements::Protect artifacts
«deriveReqt»
«deriveReqt»
«functionalRequirement»
Robot Collision Avoidance
Id = "15"
Text = "The tour robot must avoid collisions with things and people in the environment."
«performanceRequirement»
Light Illuminance
Id = "32"
Text = "The system shall not produce illuminances greater than 250 lux."
«refine»
«systemUseCase»
Steer the robot
«satisfy»
«satisfy»
«block»
Collision Avoidance
«block»
Collision Detection
«functionalRequirement»
Live Videos
Id = "14"
Text = "The system must provide live videos from all artifacts of the exhibition."
«cps-human»
«systemUseCase»
Steer the robot - human
«performanceRequirement»
Robot Distance
Id = "16"
Text = "A tour robot must run twice the distance of a path through the complete exhibition without recharge."
«trace»
«performanceRequirement»
Robot Mass
Id = "22"
Text = "The mass of a tour robot must not exceed 30 kg."
«performanceRequirement»
Resolution
Id = "14.1"
Text = "The resolution of the videos for the customer must be at least HD."
«performanceRequirement»
Server delay
Id = "14.2"
Text = "The server must deliver the videos with a delay of maximum 1 second to the clients."

Figure A.29 Example requirements diagram.

35	Copy of Record	The system must allow the virtual tour customer to run queries about the own customer record.
14.2	Server delay	The server must deliver the videos with a delay of maximum 1 second to the clients.
14.1	Resolution	The resolution of the videos for the customer must be at least HD.
17	Robot Speed	The maximum speed of a robot must be 8 km/h.
22	Robot Mass	The mass of a tour robot must not exceed 30 kg.
46	Connected Knowledge	The exhibitions' valuable knowledge is linked with other artifacts in the world increasing the overall knowledge.
47	More Visitors	The exhibition has 25% more visitors every year for the next 5 years
45	World Wide Access	People from all over the world are able to visit the exhibition.
16	Robot Distance	A tour robot must run twice the distance of a path through the complete exhibition without recharge.

Figure A.30 Example requirements table.

A requirement is shown as a rectangle with the keyword «requirement». Figure A.29 shows a simple requirements diagram. A requirements diagram is only useful to visualize top-level requirements relationships or to put a focus on very important aspects. SysML provides a table representation that is typically used for requirements. Figure A.30 shows a requirement table of the Virtual Tour system.

SysML also provides a matrix representation to visualize relationships between model elements. Figure A.31 shows the relationship between functional and non-functional requirements.

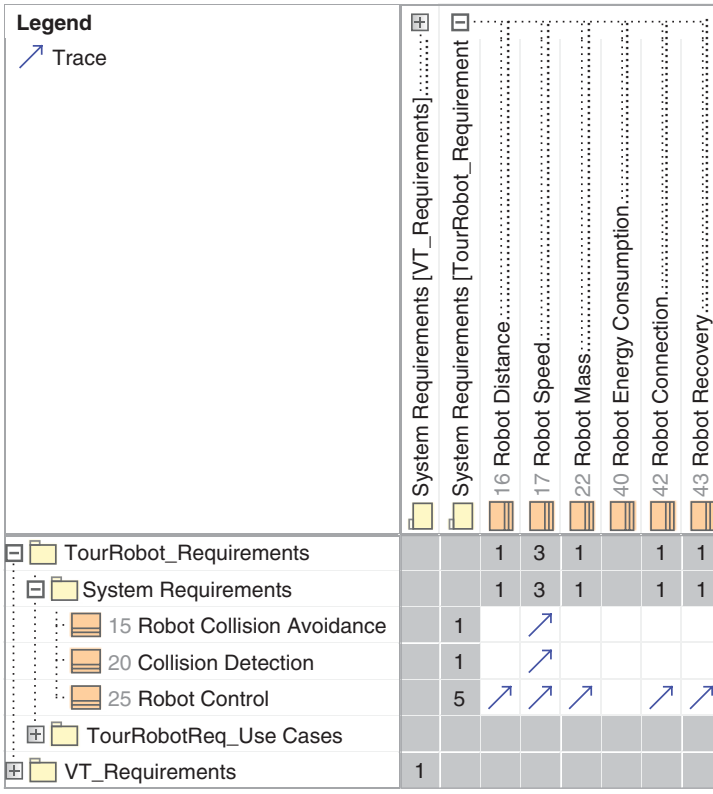


Figure A.31 Example requirements matrix.

A.6 Extension of SysML with Profiles

SysML is a common language for all types of engineered systems. Therefore, SysML provides very general model elements that are not specific for a domain or methodology. SysML can express requirements but not distinguish between functional or performance requirements. SysML can define blocks but not distinguish between hierarchy elements, such as systems, subsystems, units, modules, or segments.

In practice, you need a more specialized set of model elements for your specific purpose. SysML provides an extension mechanism to introduce new model elements to the language. You can't define new elements out of scratch. They must be based on existing SysML elements and further specify their semantic. Since SysML is also an extension of UML, the base element must be a metaclass of the

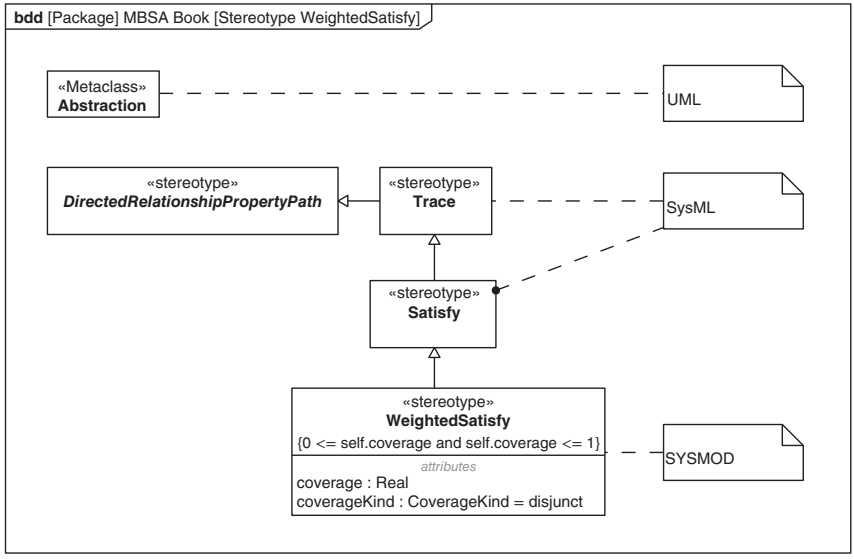


Figure A.32 Stereotype WeightedSatisfy.

UML set of model elements. Please refer to special SysML or UML literature for details, for example, [267, 269].

The model element stereotype adds new elements to the model language. It has a name and an extension relationship to the existing UML base element or a generalization relationship to a SysML or other stereotype. The stereotype can define further properties for the model element – sometimes also called tagged values – and a new icon. The semantic of the stereotype is described semiformal in natural language. Figure A.32 shows the stereotype "WeightedSatisfy" from the SYSMOD profile [267, 269]. It specializes the SysML stereotype "Satisfy" and adds a property to specify the coverage of satisfaction. For example, a system block satisfies 60% of a requirement, and another block satisfies the remaining 40%. The SysML stereotype "Satisfy" specializes the SysML stereotype "Trace" that specializes the UML stereotype "Trace" that extends the UML metaclass "Abstraction."

A profile is a special package that contains a set of stereotypes. For example, the SYSMOD profile contains all stereotypes that are useful for models created with the SYSMOD methodology [267, 269]. A profile is applied with the profile application relationship to a model enabling the usage of the profile stereotypes in that model. Figure A.19 shows the profile application of SYSMOD and SysML to the Virtual Tour system model. Note, SysML is formally also a profile of UML as described in Section A.1.

A.7 Next-Generation Modeling Language SysML v2

In December 2017, the OMG published an RFP for the next generation of SysML [240]. The RFP was preceded by many years of work to identify the requirements for a new systems modeling language that could be used effectively for the next 20+ years. The RFP contains over 200 requirements for the new SysML v2.

Since the publication of the RFP, the SysML Submission Team (SST), a consortium of about 70 organizations and almost 200 members, has been working on the new SysML v2 specification. The result will be submitted to the OMG, and, if accepted, it will be fine-tuned by a Finalization Task Force (FTF) into an official OMG specification and published by the OMG. Like SysML v1, SysML v2 could then be submitted to ISO for standardization.

SysML v2 was not yet submitted to the OMG when we handed over the work on this second edition of the book to the publisher. Therefore, SysML v2 may be different than described here or even not adopted by the OMG.

With SysML v2, you can model the same concepts as with SysML v1. That means, in particular, that you can model everything you learn in this book also with SysML v2. SysML v2 has other model elements, some of which have different names and some of which have different notations. But the similarity is very high, so you don't have to start from scratch.

A major innovation is taking place behind the scenes of SysML. While SysML v1 is based on UML, SysML v2 is defined independently of UML. It is based on its own metamodel called Kernel Modeling Language (KerML). KerML is an application-independent language for creating specific modeling languages. It can also be used to specify a UML v3, which is under discussion but not concrete yet.

SysML v2 provides many new features, for example, modeling of risks, analysis, cases, variants, and geometry, to name a few.

A special novelty is the additional textual notation. For some people and scenarios, it is much easier to work with the model using textual notation. It can also be handled by machines, for example, to automatically create SysML v2 model elements.

The following shows a shortened example of the logical tour robot architecture of the Virtual Tour system with SysML v2 textual notation, including some requirements and the satisfaction relationship.

```
package VirtualTourSystem {
  import 'Requirement Definitions'::*;

  package VT_LogicalArchitecture {
    part 'Tour Robot' {
      part engine:Engine;
      part battery {value capacity:ScalarValues::Real=42;}
      part wheel [6];
      part chassis;
      part camera;
```

```
      part 'control unit';
      value maxSpeed: ScalarValues::Real = 120;
    }
        satisfy VT_Requirements::'Tour Robot Specification' by 'Tour Robot';

    part def Engine;
  }
  package VT_Requirements {
    requirement 'Tour Robot Specification' {
      doc /* Requirement Specification of the eVehicle  */
      require tourRobotBatteryCapacity;
      require tourRobotMaxSpeed;
    }
    requirement id 'REQ.B.1' tourRobotBatteryCapacity: BatteryCapacityReqDef {
      attribute:>> capacityRequired = 50;
      attribute:>> capacityActual = VT_LogicalArchitecture::'Tour Robot'::

battery::capacity;
    }
    requirement id 'REQ.V.1' tourRobotMaxSpeed: MaxSpeedReqDef {
          attribute:>> maxSpeedRequired = 140;
          attribute:>> maxSpeedVehicle = VT_LogicalArchitecture::'Tour Robot'
          ::maxSpeed;
        }
  }
  package 'Requirement Definitions' {
    requirement def BatteryCapacityReqDef {
      doc /* The actual battery capacity shall be greater than or equal
               * to the required capacity. */
      attribute capacityActual: ScalarValues::Real;
      attribute capacityRequired: ScalarValues::Real;
      require constraint{ capacityActual <= capacityRequired }
    }
    requirement def MaxSpeedReqDef {
      doc /* The maximum speed of the robot shall be
           * not greater than the required maximum speed. */
      attribute maxSpeedVehicle: ScalarValues::Real;
      attribute maxSpeedRequired: ScalarValues::Real;
      require constraint{ maxSpeedVehicle <= maxSpeedRequired }
    }
  }
}
```

Figure A.33 depicts the same model with SysML v2 graphical notation.

Noteworthy is another new feature of SysML v2. You can directly model a part and its details without modeling the part definition. It looks similar to SysML v1. In SysML v1, it would mean to model a part property including subparts without a block. A SysML v1 block is a part definition in SysML v2, and a SysML v1 part property is similar to a part in SysML v2. Part definitions are useful to reuse definitions. But from the engineer's viewpoint, it is overmodeling to create a part definition, a part, and the relationship between them if you specify only one element without any reuse. In our SysML v2 sample model, we have modeled only an empty part definition for the engine for demonstration purposes.

Although it is possible to specify a part without a definition, SysML v2 uses the definition and usage pattern (Section 9.4) much more consistently than SysML v1. Many concepts in SysML v2 have separate definition and usage elements.

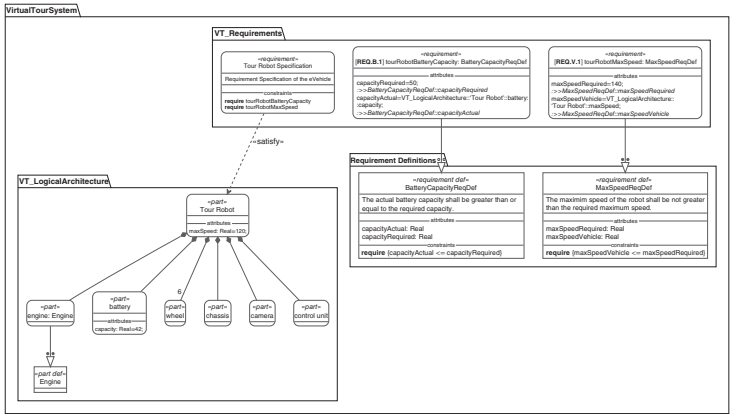


Figure A.33 Graphical notation of the SysML v2 sample model.

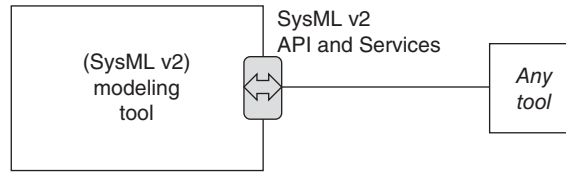


Figure A.34 SysML v2 API and Services concept.

For example, Figure A.33 depicts definitions of requirements ("BatteryCapacity ReqDef," "MaxSpeedReqDef") and usages of the definitions ("[REQ.B.1] tourRobotBatteryCapacity," "[REQ.V.1] tourRobotMaxSpeed"). SysML v1 does not apply the definition and usage pattern to requirements.

Finally, we point out another special new feature of SysML v2. The next-generation of SysML will get a SysML v2 API and Services [239]. The API provides access to a model independent of the modeling tool (Figure A.34). It could even be a non-SysML tool that provides information through the API. In the latter case, it will probably be limited to a few entities.

The SysML v2 API and Services specification will provide the implementation-independent API and Services as a PIM (see Chapter 6) and two implementation-dependent bindings as PSMs (see Chapter 6). One PSM is based on the REST/HTTP technology [77] and the other one on OSLC [193]. It is not forbidden to use other implementations, but these two are part of the official specification.

Appendix B

The V-Model

Today, the systems engineering vee is omnipresent in almost every systems development environment. It has an iconic status. The V-Model, as the systems engineering vee is also called, emphasizes a rather natural problem-solving approach. Starting on coarse grain level partitioning the problem to manageable chunks. From the fine grain level, the final solution integrates up to the initial level. On each level, one can compare solutions or part-solutions with problems or part-problems being solved. The simplicity of the V-Model (see Figure B.1) permits various projections and results in many interpretations.

This chapter provides an overview to the history of the V-Model, its intention, and considers frequent misconceptions. The chapter concludes with reading instruction of a modern V-Model.

B.1 A Brief History of the V-Model or the Systems Engineering Vee

The V-Model emerged probably in the 1960s, though there seem to be no public citations available. The citations hereinafter suggest that the V-Model independently emerged from more than one source. Designations of the model vary depending on the sources. Hereinafter, we cite designations as used in the referenced documents.

In 1979, Barry W. Boehm published a paper [30] that was built up on the vee. He used the vee in the context of software engineering to emphasize the importance of verification and validation. Boehm made a distinction between an upper part of the vee for validation and a lower part of the vee for verification and linked these processes to the related requirements and specifications, respectively. He did not elaborate on the multilevel nature of systems. Boehm attributed the "V-chart," as he named it, to personal communication from J.B. Munson, System Development Corporation in 1977.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

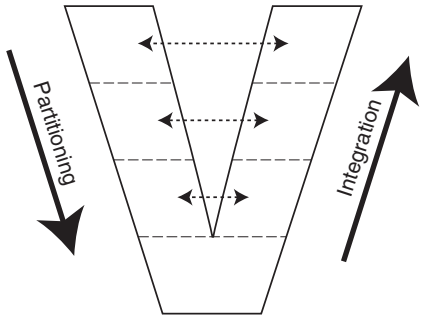


Figure B.1 A basic V-Model.

In 1986, Paul Rook depicted in his article "Controlling software projects" the software life cycle in a v-shape and named it "V-diagram" [210]. The described software life cycle considers multiple levels, decomposition, and integration aspects of a V-Model as described in later literature.

In a systems engineering context, the V-Model was presented at the first annual conference of NCOSE in 1991 [79]. This conference was the predecessor of today's INCOSE International Symposium. Kevin Forsberg and Harold Mooz introduced the "'Vee' model" to clarify the role and responsibility of system and design engineering within the cycle of a project. Different than Boehm's paper that focused on V&V processes in software engineering, Forsberg and Mooz focused on projects realizing systems. Consequently, the multilevel nature of systems is a major issue within that paper. They presented a three-dimensional "'Vee' model" intending to explain the relations between project management and engineering. It acknowledges the iterative and incremental nature of engineering and hence encourages concurrent engineering. Forsberg and Mooz stated to follow the basic outlines of the "'Vee' chart" developed by NASA for software management and assurance. They mentioned a major contribution to their "'Vee' chart" by Richard Roy.

Kevin Forsberg and Hal Mooz together with Howard Cotterman further elaborated on "Vee Models" in "Visualizing Project Management" [80].

In 1992, Germany published a standard for governmental IT projects. The standard originated from technology projects by the German ministry of defense in 1986 [37]. The goals of these technology projects were optimizing cost and quality of software developments, decoupling acquirer and supplier processes, and increasing transparency of software developments. The resulting comprehensive standard is a process description designated as "V-Modell®." The vee did not stand for the graphical representation of the process model. "V-Modell®" was an abbreviation for the German term "Vorgehensmodell" what could be translated as "process model" [37, 51]. Nevertheless, the described processes are depicted in a v-shape. The 1992 edition covered only software development. It was made available to the public by Bröhl and Dröschel in "Das V-Modell" [37] in 1993. A further developed edition of the "V-Modell®," published in 1997,

included systems aspects with references to the system definition by ISO 12207. The standard evolved and became the "V-Modell® XT" with the XT standing for extendable or extreme tailoring. The latest description of the "V-Modell® XT" does no longer explicitly define the meaning of the vee. The questions and answers section relates the vee to the idea of displaying the specification decomposition and solution integration face to face [259].

The German engineering association "VDI" (abbreviation for the German name "Verein Deutscher Ingenieure") developed the guideline VDI 2206:2004 [257] describing a method for developing mechatronic systems which is frequently related to the V-Model. VDI 2206:2004 considers the process model described in the "V-Modell®" mentioned above as not sufficient for mechatronic developments. The guideline emphasizes the need of mechatronic developments to mature certain parts faster that the system-of-interest to solve cross-domain issues early in the development cycle. Consequently, the described method includes incremental and recursive approaches. VDI 2206:2004 shows similarities with the V-Model paper by Forsberg and Mooz [79].

At the INCOSE International Symposium 2013, Dieter Scheithauer and Kevin Forsberg presented the paper "V-Model Views" [214]. This paper collects experiences and improvements from the preceding two decades. The authors extend the scope of the V-Model from the development process to the life cycle of the system, reaching from stakeholders' needs to stakeholders' satisfaction. Compared to the V-Model presented 1991, Scheithauer and Forsberg split the overall view into four distinct views: Basic-V, Development-V, Assurance-V, and Dynamic-V. The horizontal dimension evolves from a time or maturity sequence to a logical sequence of the value stream. Scheithauer and Forsberg emphasize the inductive design of left side of the vee in opposition to a deductive decomposition imposed by a waterfall-like process. And they make explicit that validation does not only apply for the finally deployed system but also to each artifact along the life cycle of the system. That is, stakeholder requirements, system requirements, architecture on each level and integrated into the whole system architecture, as well as the system in operation need to be validated. Validation means checking fitness for purpose of the mentioned artifacts. The authors mention in the paper that the V-Model had been introduced twice, in the 1980s by NASA and with the paper by Forsberg and Mooz in 1991.

B.2 A Handy Illustration but No Comprehensive Process Description

The systems engineering vee is an illustration that depicts only some aspects of the development process of a system. Apart from the German "V-Modell®," the systems engineering vee is no comprehensive process model or process

description. Probably, the most important aspect depicted in the vee is the multilevel nature of systems. A system needs to comprise at least two levels (see Section 4.1). Therefore, the simplest vee considering the system-of-interest only would comprise two levels, the system level and the system element level.

The levels of the vee correlate with the system decomposition levels. This includes also logical levels. The designation of these levels varies depending on the context of application. But each consecutive level pair represents a system in its own right. The lowest level is a bit special, as parts or components represented by this level will not further be partitioned from the viewpoint of the depicted vee. This imposes that elements represented by the lowest level can be acquired, produced, or harvested. The compositions of these elements do not matter in respect to the system-of-interest. This does not preclude the lowest level elements to be systems viewed from a different viewpoint. As with each model, the extent of a V-Model should follow a purpose. It typically includes the levels for which a certain team or organization is responsible adding the adjacent lower level and sometimes the adjacent upper level.

Depicting development-related life cycle processes of systems within the vee, each level needs to comprise instances of the same processes. On each level requirements engineering, architecting, integration, and verification need to be performed. The concrete naming of these processes will vary depending on the organization applying these processes. These processes will exhibit the zigzag pattern as described in Section 9.1. Figure B.2 depicts a vee with the basic development processes creating artifacts as named in Figure 9.1. It displays that the zigzag pattern not only exists on the left side of the vee but proceeds up the right side as well while integrating system elements to systems.

The dotted lines in Figure B.2 designate processes not in the responsibility of the assumed development. The system levels addressed in the vee in are from top:

- The business as system to earn money.
- A stakeholder process gaining benefit from using the product to be developed.

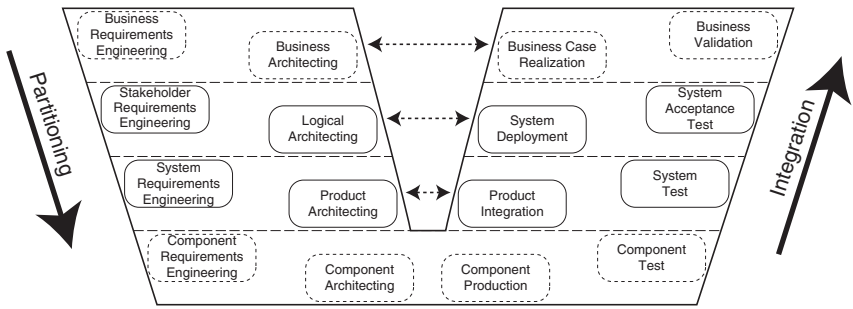


Figure B.2 V-Model with exemplary named basic development processes.

- The product to be developed.
- Components from which the product is built.

Figure B.2 does not serve as a development process description. Many important elements are elided. It does not depict any control or object flows. It only maps processes to system levels. The figure designates a requirement engineering process but does not make any distinction between stakeholder and system requirements. In absence of any object flows, it makes no distinction between allocated requirements and elicited requirements. The integration processes are shown with a variety of names, and verification and validation processes are mixed and partly designated as test. Figure B.2 follows here a widely used, but rather suboptimal naming by mixing various topics. Test is used as synonym for verification and validation. It imposes test as a generalized method covering other methods such as inspection, analysis, or demonstration. And validation is reduced to an activity at the top right end, though it should start as early as during the stakeholder requirements definition process. Ideally, a V-Model representation should focus on a dedicated aspect. That is, many V-Models exist, each for a specific purpose [214].

The rather simple illustration depicted above neglects a further issue. The number of elements on lower levels is higher than on higher levels. Reusing elements in modular systems will manage the increase of element kinds on subsequent levels. Each of these elements has its own life cycle. Their development or integration sequence depends on availability of data from the adjacent elements. For a comprehensive illustration representing the multiplicity of element kinds, a three-dimensional vee needs to be drawn. This third dimension would consider the interaction between the processes related to different elements on the same level. One can easily imagine, that an illustration with a three-dimensional vee as depicted in Figure B.3 enriched with processes as depicted above becomes difficult to read and understand. Considering the elided items mentioned above would make understanding even worse.

B.3 Critical Considerations

The V-Model received many interpretations. Many of them did not consider the original purpose and criticized the model for not fitting another purpose. The world evolved during the decades since the introduction of the V-Model. New or refined engineering methods came up, new technology, and tools provide possibilities to improve effectivity and efficiency of our work. The V-Model can still help to illustrate the multilevel nature of systems and dependencies during the development process. But this can hardly be illustrated in a single view. Dieter Scheithauer and Kevin Forsberg provided with their paper

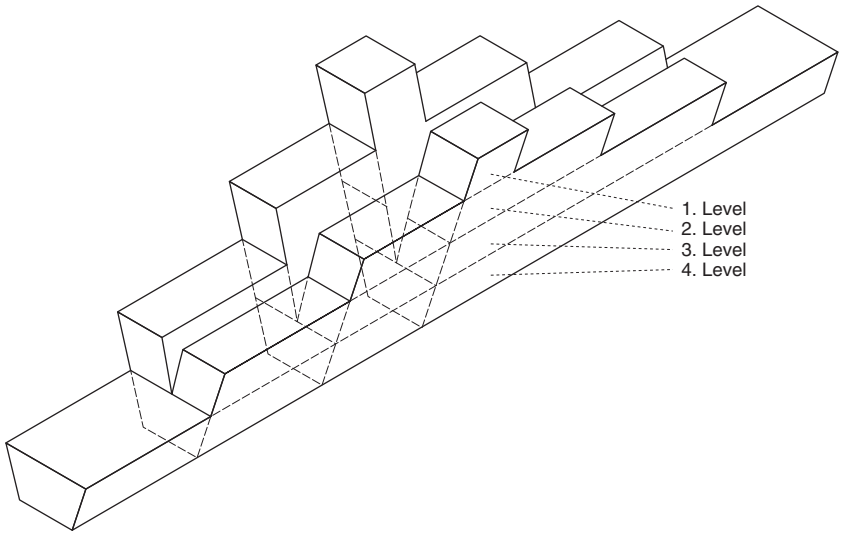


Figure B.3 V-Model considering discrete levels and numbers of elements.

"V-Model Views" [214] explanations and summary for appropriate application of the V-Model. We address hereinafter some issues that frequently leading to discussions.

B.3.1 The V-Model as Process Description

The systems engineering vee, unlike the German "V-Modell®," was never intended as a comprehensive process description. It is intended to describe the multiple levels of a system and only some specific process aspects related to such level and their dependencies. The V-Model can assist in explaining that the life cycle processes need to be applied at each level and for each element belonging to these levels.

B.3.2 The V-Model Does Not Impose a Waterfall Process

Interpreting the V-Model as equal or similar to a waterfall approach, neglects the meaning of levels within the V-Model. In a waterfall approach each process, such as requirements engineering, architecting, implementation, verification, and final validation is considered on its own level. A waterfall process is intended to flow once from the top (requirements engineering) down to the bottom (final validation). When considering the V-Model as waterfall, the second half of the waterfall cascade is bended up to shape a vee. This results in requirements engineering and final validation appearing on the top level, followed by architecting and verification on the intermediate level and implementation on the tip of the vee. This view

neglects the meaning of the levels to represent the decomposition levels of the system-of-interest.

Unlike with a waterfall, the horizontal dimension of the vee represents a logical sequence in the value stream and not a time line of the development process. Allocating requirements to lower levels and subsequently validating these requirements inevitably demands communication. Such communication ensures the bidirectional data exchange. Considering the life cycle of each level and hence applying recursively the life cycle processes at each level results in a distinction between allocated requirements and other stakeholder requirements. The upper level becomes only one among other stakeholders of the lower level. Unlike with a waterfall approach, where all requirements are supposed to be available up front, the V-Model permits and requires induction of additional stakeholder requirements at lower levels. This permits to reinitiate interactions with stakeholder to clarify or capture previously missing requirements.

B.3.3 The V-Model Accommodates Iterations

Maintaining a consistent configuration baseline on each level enables iterations in several forms. Iterations are possible on system element level on either side of the vee. They can involve one or more levels. But also the big iterations including both sides over one or more levels are possible. Especially, the small iteration loops build up on early validation, such as checking requirements allocation and design of system element for their fitness to contribute to the overall goal. Virtual integration of such system elements in the system model permits validation of each system level and each system element before their implementation starts. Iterative approach in the development of systems is not new and had been addressed already by the invention of Scrum [241]. Forsberg and Mooz promoted iterative development in their 1991-paper [79]. The same applies for early validation of artifacts. Though not described in detail, Boehm mentioned in his 1979-paper [30] as example requirements validation, design validation, and validation tests for the final software. He did already emphasize the benefits of early validation.

B.3.4 The V-Model Permits Incremental Development

Using system models and applying iterations as explained above permits incremental approaches. The use of system models permits simulations and demonstration of the virtual integrated system. This permits early validation of the system with the related stakeholders. Incremental development was already promoted by Forsberg and Mooz in 1991 [79].

B.3.5 The V-Model and Concurrent Engineering

Since the V-Model emphasizes system levels, each with a number of system elements with their life cycle, it can be used to explain impact of concurrent engineering. Maintaining a consistent configuration baseline and defining increments for each iteration enables concurrent engineering.

B.3.6 The V-Model Accommodates Change

An incremental and iterative approach and the maintenance of a consistent configuration baseline permits to predict impact of injected changes. The V-Model can assist in explaining where change can be expected. And it can visualize where these changes impact the development. Sources of change include discovering of new stakeholders, changing stakeholder needs, or error detection at each integration, validation, or verification step.

B.3.7 The V-Model Permits Early Verification Planning

An incremental and iterative approach and the maintenance of a consistent configuration baseline permits early verification planning. Consequently, optimization of the verification processes and related infrastructure can be achieved. This includes combining verifications of different levels or combining interface verification during integration with system or system element verification.

B.3.8 The V-Model Shows Where to Prevent Dissatisfaction

The V-Model can display where validation should be performed. Validation does not only apply to the top-right most part of the systems engineering vee. Each artifact created during a development should be validated. That is, evidence should be provided that the considered artifact contributes to satisfy imposed requirements. Early validation ensures that the right problem is solved. The standard ISO/IEC/IEEE 15288:2015 [115] requires in the quality assurance process validation of outputs of the life cycle processes. Such early validation are sometimes named as “in-process validation” or “continuous validation” [99, 265]. The standard ISO/IEC/IEEE 29148:2018 [117] emphasizes the necessity of requirements validation. Scheithauer and Forsberg make the execution of validation very explicit in their Assurance-V [214]. Validation, and hence prevention of dissatisfaction starts at the top-left most part of the V-Model and proceeds all the way down and up again, even beyond to the operation phase of the

system-of-interest. Scheithauer and Forsberg identified six kinds of validations along a system life cycle:

1. Validation of stakeholder requirements
2. Validation of allocated requirements
3. Validation of system element definitions
4. Validation of the virtually integrated system
5. Validation of the operational system deployed into its environment
6. Validation of the in-service system

B.4 Reading Instruction for a Modern Systems Engineering Vee

To summarize this chapter, we provide a reading instruction for V-Models. As mentioned earlier, a single view can hardly depict each aspect. Views have to be designed to frame requirements or concern of dedicated stakeholders and elide other aspects. Following the seven rules stated hereinafter will ensure a consistent understanding.

B.4.1 The Vertical Dimension

The vertical dimension of the V-Model denotes the multiple levels of the system-of-interest. The top most level represents the system context in which the system-of-interest operates. The bottom most level represents the parts that can be obtained. Consequently, representing parts in black-box-views is sufficient. The levels in between represent product or logical levels of the system. Therefore, the V-Model should comprises of at least three levels, system context, system-of-interest and system elements.

B.4.2 The Horizontal Dimension

The horizontal dimension of the V-Model denotes a logical sequence of the value stream. This does not impose that each requirement needs to be frozen upfront. An increment in development may impose to develop some parts before defining the remaining requirements on top level.

B.4.3 The Left Side

The left side of the V-Model denotes the general direction of the top-down development. This is only the general direction. Especially, interface related issues or

technology scouting require to push parts of the development to very low levels before proceeding with the remaining parts of the upper levels.

B.4.4 The Right Side

The right side of the V-Model denotes the general direction of the bottom-up integration. A model-based approach enables virtual integration. This results in a number of instances of the right side. Such representation are sometimes called Y-Model as such virtual integration can be depict with a branch starting in the middle of the left side and heading upwards in parallel to the right side of the vee.

B.4.5 The Levels

The Levels represent the system decomposition levels. Life cycle processes are applied on each level. Levels get requirements allocated from the next level above, elicit requirements from its specific stakeholders, and allocate requirements to the next level below. Levels receive verified system elements (or parts of) from the next level below and provide the integrated and verified system (or parts of) to the next level above.

B.4.6 Life Cycle Processes

The life cycle processes such as stakeholder requirements definition, system requirements definition, architecture definition, design definition, integration, verification, validation, and the related artifacts appear on each level.

B.4.7 The Third Dimension

The third dimension of the V-Model can be used for different purposes. For instance to display, the multitude of system elements per level resulting in parallel application of the life cycle processes.

Appendix C

Glossary

Many definitions in this book are model-based and depicted in SysML to maintain relations and coherency. Still a glossary can provide benefit following the "principle of definition" asking for both formal definition for precision and prose for comprehensibility [56].

C.1 Heritage of the Term "Glossary"

A glossary is an alphabetically ordered list of terms with related explanations. The term derives from the medieval Latin "glossa" meaning explanation of a difficult word. It traces to the Greek "glōssa" standing for "word needing explanation" [196]. Glosses were added in medieval books marginal or interlinear as depicted in Figure C.1.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

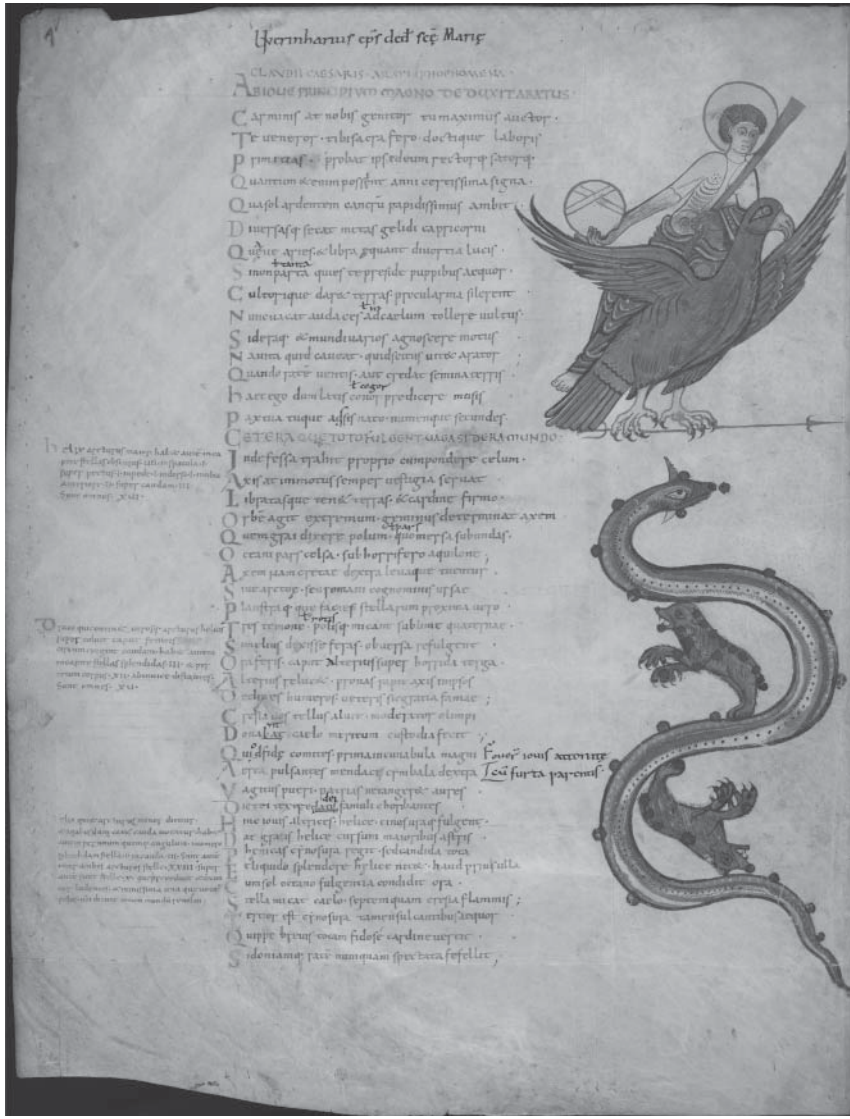


Figure C.1 A page of an eleventh century manuscript with glosses. Source: Reproduced with permission from Burgerbibliothek Bern, Cod. 88, f. 1v – Aratus – Germanicus: Phaenomena. (https://www.e-codices.ch/en/list/one/bbb/0088)

C.2 Terms with Specific Meaning

Term	Definition
actor	An actor represents a role outside the system-of-interest and can be a user or another system.
architecture	Architecture is an idea of a system in its context embodied in its elements, interactions among its elements and context, and the principles guiding the system's organization, design and evolution, and the decisions along its development and evolution [56, 114, 266].
architecture decision	An architecture decision traces a step in the evolution of architecture by documenting why architecture came to its existence [114].
architecture description	An architecture description is a work product used to express architecture [114].
architecture description element	An architecture description element is a building block used to represent what, how, or why (including why not) of architecture solving problems [114].
architecture description language	An architecture description language is any system of communication for use in architecture descriptions [114, 196].
architecture framework	An architecture framework defines conventions, principles, and practices for the description of architectures established within a specific domain of application or community of stakeholders [114].
architecture rationale	An architecture rationale records explanations, justifications, or reasoning about architecture decisions that have been made [114].
architecture view	An architecture view is a work product visualizing architecture from a specific architecture viewpoint addressing requirements or concerns [114].
architecture viewpoint	An architecture viewpoint is a work product establishing the conventions for the construction, interpretation, and use of architecture views to frame specific requirements or concerns [114].
base architecture	A base architecture is an imposed solution on any level of abstraction.
concern	A concern denotes an interest of any kind into the system-of-interest.
constituent system	A constituent system (CS) is the role played by systems being constituents of systems of systems.

(*continued*)

Term	Definition
constraint requirement	A constraint requirement specifies a condition not needing to be implemented, but must be complied with.
context interaction	A context interaction is an exchange of matter, energy, data, or combinations thereof between the system-of-interest and its system context [197].
core activity	A core activity defines a functionality that does not directly involve an actor.
core group	A core group collects use case activities of the system-of-interest without actor interactions.
cyber-physical system	A cyber-physical system (CPS) is an engineered system converging physical systems and cyber systems tightly integrating computing, communication, and control technologies with possibly interacting humans.
cyber-physical system of systems	A cyber-physical system of systems (CPSoS) comprise of constituent systems that are managerially or operationally independent cyber-physical systems.
design principle	A design principle is a generally accepted approach guiding the design of an entity [196].
digital engineering	Digital engineering is an integrated digital approach using authoritative sources of systems' data and models as a continuum across disciplines to support life cycle activities from concept through disposal [54].
discipline	A discipline identifies a branch of knowledge and field of studies or training [196].
engineering discipline	An engineering discipline is concerned with requirements, architecture, design, implementation, integration, deployment, use, maintenance, and disposal of engines, machines, and structures [196].
entity	An entity is a thing with a distinct and independent existence [196].
evolution principle	An evolution principle is a generally accepted approach guiding the evolution of an entity [56, 114, 196].
functional architecture	A functional architecture bases on functional elements, functional interfaces, and architecture decisions. Functional architecture is based on a hierarchical function structure resulting from functional decomposition.
functional connection	A functional connection is between functional interfaces defining a functional flow of information, materials, force, or energy.
functional element	A functional element abstracts a set of functions and defines functional interfaces specifying functional (object) flows to other functional elements.

Term	Definition
functional group	A functional group collects a set of strongly related use case activities.
functional interaction point	A functional interaction point collects a set of input and output parameters of functional elements.
functional requirement	A functional requirement specifies functionality that the system-of-interest shall provide.
I/O activity	An I/O activity defines a functionality of the system-of-interest implementing exchange of items with actors.
I/O group	An I/O group collects use case activities specifying interactions between actors and the system-of-interest.
interaction principle	An interaction principle is a generally accepted approach guiding the design of interactions between entities [196].
leaf entity	A leaf entity is a system element that can be acquired. Therefore, a leaf entity can be considered in a black box view only and does not need further decomposition in the context of the system-of-interest. Declaring system elements as leaf entities is a subjective decision depending on the respective viewpoint.
logical architecture	A logical architecture comprises solution principles and is not yet as concrete as to be considered as a product.
model-based engineering	Model-based engineering (MBE) is a kind of digital engineering with focus on models. MBE is seen as the resulting concept of combining life cycle spanning management of product data (PLM) and formal description of systems (MBSE) [90].
model-based requirements engineering	Model-based requirements engineering (MBRE) is the formal application of modeling to support requirements engineering activities.
model-based systems architecting	Model-based systems architecting (MBSA) is the formal application of modeling to support systems architecting activities.
model-based systems engineering	Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [122].
organization principle	An organization principle is a generally accepted approach guiding the organization of entities within systems [56, 114, 196].
perspective	A perspective is a way of perceiving architecture focusing to specific facets.

(*continued*)

Term	Definition
physical architecture	A physical architecture comprises physical elements, design principles, and architecture decisions. Physical elements are hierarchically structured as result of physical decomposition permitting allocation of functional elements and functional connections.
product architecture	A product architecture is a solution comprising of concrete product elements. The elements do not need to be defined as concrete to be producible.
purpose	A purpose defines what a stakeholder intends to achieve by using a system [6, 114].
quality requirement	A quality requirement specifies quality features that are not covered by functional requirements.
requirement	A requirement specifies a capability or condition that must or should be satisfied.
requirements specification	A requirement specification is a collection of requirements.
stakeholder	A stakeholder is an individual or organization having interests in a system [115].
stakeholder process	A stakeholder process provides benefit or detriment in any way to the involved stakeholder [115, 227, 265].
system (engineered)	An engineered system is a composition of multiple system elements. Its existence is justified by purposes and it exposes system characteristics permitting stakeholders to achieve purposes. Systems are situated in their system context and may be evolved.
system architecture	A system architecture is a composition of functional architecture, logical architecture, and product architecture.
system behavior	A system behavior is an observable response by the system-of-interest to a stimulation. The stimulation can originate from the system context or from a system element [196].
system boundary	A system boundary is the line of demarcation between the system-of-interest and its system context [265].
system characteristic	A system characteristic is a distinguishing quality explaining what a system is or does [196].
system context	A system context comprises the system-of-interest and each element outside the system boundary that interacts with the system-of-interest in a non-negligible way [114, 196].
system design	A system design is a concretized system architecture realizing the involved discipline specific architectures. The identification of and function allocations to system elements are as concrete as to enable system elements' implementation.

Term	Definition
system element	System element is the role played by entities being constituents of systems [115, 265].
system element interaction	A system element interaction is an exchange of matter, energy, data, or a combination thereof to realize system characteristics [197].
system evolution	A system evolution is a gradual development of systems over time [21, 196].
system meaning	A system meaning denotes a significance of the system-of-interest. It relates to a constructionist perspective on the system-of-interest [227].
system of systems	A system of systems (SoS) comprise of constituent systems that are managerially or operationally independent systems.
system property	A system property is an observable or measurable quality attribute of the system-of-interest.
system-of-interest	System-of-interest (SoI) is the role played by the system under consideration [115, 265].
systems engineering	Systems engineering (SE) is a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods [227].
use case	A use case specifies a coherent interaction of actors with the system-of-interest. The interaction is initiated by a trigger from the outside and ends with a result that is of value for the actor or other stakeholders of the system-of-interest.
use case activity	A use case activity defines a requested system function and can be further decomposed by other called use case activities (functional decomposition).
visualization	A visualization presents architecture description elements according to conventions of the governing visualization kind.
visualization kind	A visualization kind collects conventions on how to visualize architecture description elements.

References

1 www.cisecurity.org, visited 9 August 2021.
2 J. A. Zachman. About the Zachman Framework. https://www.zachman.com/about-the-zachman-framework, accessed 16 February 2015.
3 J. Abulawi. Personal communication. February 12, 2014.
4 R. L. Ackoff. Towards a System of Systems Concepts. *Management Science*, 17(11):661–671, 1971.
5 R. L. Ackoff. *Creating the Corporate Future: Plan or be Planned For*, Copyright (C) 1981 by John Wiley & Sons, Inc.
6 R. L. Ackoff. *Differences That Make a Difference*. Triarchy Press Ltd, 2010.
7 S. Ackva, C. Lalitsch-Schneider, M. Andres, S. Husung, M. Leute, and A. Haarer. Modellbasierter Bärentango mit dem System Architecture Framework. In *Tag des Systems Engineering (online)*, November 2021.
8 S. Ackva and M. Leute. System Architecture Framework als Werkzeug zur MBSE Einführung. In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and S. Ackva, editors, *Tag des Systems Engineering, Paderborn, November 2017*, pages 259–262. München, Germany, 2017. Carl Hanser Verlag.
9 S. Aeschbacher, S. Eisenring, D. Endler, M. Frikart, N. Krüger, J. G. Lamm, and M. Walker. Eine einfache Vorlage zur Architekturdokumentation. In M. Maurer and S.-O. Schulze, editors, *Tag des Systems Engineering, Bremen, 12–14 November 2014*, pages 85–92. München, Germany, 2014. Carl Hanser Verlag.
10 AFIS. *Systems Product Line Engineering Handbook*. AFIS.
11 G. Akhras. Smart materials and smart systems for the future. *Canadian Military Journal*, 1(3):25–31, 2000.
12 A. Albers, G. Moeser, and M. Salihoglu. Meshing free sketches in SysML system models to support design engineering, 2017.
13 C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, MA, 1971 (paperback reprint of the 1964 edition).
14 C. Alexander. *A Pattern Language*. Oxford University Press, New York, 1977.

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

15 T. J. Allen. *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information Within the R&D Organization*. Massachusetts Institute of Technology, 1st edition, 1977. Now in a revised edition, 1984 from MIT Press.

16 D. Allen. *Getting Things Done*. Peguin Books, 2001.

17 G. Altshuller. *Innovation Algorithm: TRIZ, Systematic Innovation and Technical Creativity*. Technical Innovation Center, 1999.

18 S. W. Ambler. *Agile Modeling*. John Wiley & Sons, 2002.

19 M. M. Andreasen. *Syntesemetoder på systemgrundlag (Machine Design Methods Based on a Systematic Approach)*. PhD thesis, Lund Technical University, Sweden, 1980.

20 Automated Instrument for Diagrams (AID) on sourceforge.net. https://sourceforge.net/projects/aid4md/, accessed 6 June 2021.

21 J. Baedke and S. F. Gilbert. Evolution and development. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2020 edition, 2020.

22 V. R. Basili and C. Larman. Iterative and incremental development: a brief history. *Computer (IEEE Computer Society)* 36:47–56.

23 L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

24 R. Batista, A. Haarer, S. Ackva, and C. Lalitsch-Schneider. System architecture framework (SAF). In *EMEASEC/TdSE 2018*.

25 E. N. Baylin. *Functional Modeling of Systems*. Gordon and Breach, New York, 1990.

26 R. Beasley, I. Cardow, M. Hartley, and A. Pickard. Structuring requirements in standard templates. In J. Lalk, editor, *EMEASEC 2014 27 – 30 October 2014, Systems Engineering – Exploring New Horizons*. INCOSE, 2014.

27 B. J. L. Berry. Cities as systems within systems of cities. In *Papers and Proceedings of the Regional Science Association*, volume 13, pages 147–163. Springer, 1964.

28 B. S. Blanchard. *System Engineering Management*. John Wiley & Sons, 2004.

29 H. Blume, H. T. Feldkaemper, and T. G. Noll. Model-based exploration of the design space for heterogeneous systems on chip. *Journal of VLSI Signal Processing*, 40:19–34, 2005.

30 B. W. Boehm. Guidelines for verifying and validating software requirements and design specification. In *Proceedings of the European Conference on Applied Information Technology of the International Federation for Information Processing (Euro IFIP)*, volume 1, pages 711–719, 1979.

31 B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

32 A. Bondavalli, S. Bouchenak, and H. Kopetz, editors. *Cyber-Physical Systems of Systems - Foundations - A Conceptual Model and Some Derivations: The AMADEOS Legacy*, volume 10099 of *Lecture Notes in Computer Science*. Springer, 2016.

33 D. A. Bonebright. 40 years of storming: a historical review of Tuckman's model of small group development. *Human Resource Development International*, 13(1):111–120, 2010.

34 J. M. Borky and T. H. Bradley. *Effective Model-Based Systems Engineering*. Springer, 2019.

35 K. E. Boulding. General Systems Theory—The Skeleton of Science. *Management Science*, 2(3):197–208, 1956.

36 Tate Britain (Tate Gallery). Press Release – Jimmy Wales announces robots project as the winner of first IK Prize. http://www.tate.org.uk/about/press-office/press-releases/jimmy-wales-announces-robots-project-winner-first-ik-prize, published February 7th, 2014, accessed August 14th, 2014.

37 A.-P. Bröhl and W. Dröschel, editors. *Das V-Modell*. R. Oldenbourg Verlag, München, Germany / Wien, Austria, 1993.

38 D. M. Buede. *The Engineering Design Of Systems*. John Wiley & Sons, 2009.

39 H.-J. Bullinger and A.-W. Scheer. *Service Engineering – Entwicklung und Gestaltung innovativer Dienstleistungen*, pages 3–18. Springer, 2006.

40 B. Buxton. *Sketching User Experiences*. MITP, 2014.

41 E. Chang. *Exploring Digital Technologies for Art-Based Special Education: Models and Methods for the Inclusive K–12 Classroom*, chapter Museums for Everyone, pages 65–76. Routledge, 2019.

42 P. Cilliers. *Complexity and Postmodernism: Understanding Complex Systems*. Routledge, London, 1998.

43 P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 3rd edition, 2001.

44 R. J. Cloutier and D. Verma. Applying the concept of patterns to systems architecture. *Systems Engineering*, 10:138–154, 2007. https://doi.org/10.1002/sys.20066.

45 A. Cockburn. *Crystal Clear: A Human-powered Methodology for Small Teams*. Addison-Wesley, 2005.

46 J. M. Cohen and M. J. Cohen. *The Penguin Dictionary of Twentieth-Century Quotations*. VIKING / Penguin Books, 1993.

47 Command Architecture Capability Team (ACaT) of the NATO Consultation and Control Board (C3B). NATO Architecture Framework Version 4. https://www.nato.int/nato_static_fl2014/assets/pdf/2021/1/pdf/NAFv4_2020.09.pdf, accessed 27 September 2021.

48 M. Conrad. Systematic testing of embedded automotive software - the classification-tree method for embedded systems (CTM/ES). In E. Brinksma,

W. Grieskamp, and J. Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

49 L. L. Constantine. Segmentation and design strategies for modular programming. In T. O. Barnett and L. L. Constantine, editors, *Modular Programming: Proceedings of a National Symposium*. Information & Systems Press, 1968.

50 E. F. Crawley, B. G. Cameron, and D. Selva. *System Architecture: Strategy and Product Development for Complex Systems*. Pearson, 2016.

51 Das V-Modell®. http://v-modell.iabg.de, accessed 8 November 2014. Still accessible at https://web.archive.org/web/20210308172827/https://v-modell.iabg.de/, accessed 19 June 2021.

52 J. S. Dahmann and K. J. Baldwin. Understanding the Current State of US Defense Systems of Systems and the Implications for Systems Engineering. In *2008 2nd Annual IEEE Systems Conference*, pages 1–7. IEEE, 2008.

53 J. S. Dahmann, G. Rebovich Jr., and J. A. Lane. Systems Engineering for Capabilities. *CrossTalk (The Journal of Defense Software Engineering)*, 21(11):4–9, 2008.

54 Defense Acquisition University. Definition: digital engineering. https://www.dau.edu/glossary/Pages/Glossary.aspx#!both|D|27345, accessed 7 July 2021.

55 J. Dick and J. Chard. The systems engineering sandwich: combining requirements, models, and design. In *Proceedings of the Fourteenth Annual International Symposium of the International Council on Systems Engineering*, 2004.

56 C. E. Dickerson and D. N. Mavris. *Architecture and Principles of Systems Engineering. Complex and Enterprise Systems Engineering*. Auerbach Publications Taylor & Francis Group, 2010.

57 K. Dolan, S. Prince, S. Dixon-Fyle, and V. Hunt. Diversity wins: how inclusion matters.

58 D. Dori and H. Sillitto. What is a system? An ontological framework. *Systems Engineering*, 20:207–219, 2017.

59 N. R. Draper and G. E. P. Box. *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons, New York, 1987.

60 Y. Duan, Y. Cao, and X. Sun. Various "aaS" of everything as a service. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 01 June 2015 – 03 June 2015.

61 M. Dänzer, W. Gerritsen, J. G. Lamm, and T. Weilkiens. Funktionale Architektur trifft Schichtenarchitektur. In M. Maurer and S.-O. Schulze,

editors, *Tag des Systems Engineering, Paderborn, 7.-9. November 2012*, pages 363–372. München, Germany, 2012. Carl Hanser Verlag.

62 M. Dänzer, S. Kleiner, J. G. Lamm, G. Moeser, F. Morant, F. Munker, and T. Weilkiens. Funktionale Systemmodellierung nach der FAS-Methode: Auswertung von vier Industrieprojekten. In M. Maurer and S.-O. Schulze, editors,*Tag des Systems Engineering, Bremen, 12.-14. November 2014*, pages 109–118. München, Germany, 2014. Carl Hanser Verlag.

63 A. Edmondson. Psychological safety and learning behavior in work teams. *SAGE Administrative Science Quarterly (ASQ)* 44:350–383.

64 O. C. Eichmann, J. G. Lamm, S. Melzer, T. Weilkiens, and R. God. CPS-FAS Method: Deriving functional architectures for cyber-physical systems using interconnectable models. In *TdSE – Tag des Systems Engineering*, 2020.

65 S. Eisenring, M. Frikart, W. Gerritsen, C. Krainer, J. G. Lamm, A. Mettauer, M. Walker, and M. Zollinger. Auf dem Weg zu einem Leitfaden im Systems Engineering für moderat-komplexe Systeme. In M. Maurer and S.-O. Schulze, editors, *Tag des Systems Engineering, Paderborn, 7.-9. November 2012*, pages 33–42. München, Germany, 2012. Carl Hanser Verlag.

66 H. Eisner. *Managing Complex Systems: Thinking Outside the Box*. Hoboken, N.J.: John Wiley & Sons, 2005.

67 H. Eisner, R. McMillan, J. Marciniak, and W. Pragluski. RCASSE: Rapid Computer-Aided System of Systems (S2) Engineering. In *INCOSE International Symposium*, volume 3, pages 267–273. Wiley Online Library, 1993.

68 J. P. Elm and D. R. Goldenson. The business case for systems engineering study: results of the systems engineering effectiveness survey. Technical Report CMU/SEI-2012-SR-009, Software Engineering Institute, 2012.

69 M. Emes. Strategic multi-stakeholder trade studies. In *Proceedings of EuSEC*, 2006.

70 M. R. Emes, P. A. Bryant, M. K. Wilkinson, P. King, A. M. James, and S. Arnold. Interpreting "systems architecting". *Systems Engineering*, 15(4):369–395, 2012.

71 D. Endler, A. Haarer, C. Lalitsch-Schneider, M. Krammer, H. Martin, T. Weilkiens, and A. Berres. System safety in SysML. In S. O. Schulze, C. Christian, editors *Tag des Systems Engineering 2015*. Carl Hanser Verlag, München.

72 S. Engell, R. Paulen, M. A. Reniers, C. Sonntag, and H. Thompson. Core Research and Innovation Areas in Cyber-Physical Systems of Systems. In *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, pages 40–55, 2015.

73 Federal Enterprise Architecture (FEA). http://www.whitehouse.gov/omb/e-gov/fea, accessed 17 February 2015.

74 Federation of EA Professional Organizations (FEAPO). Common perspectives on enterprise architecture. *Architecture and Governance Magazine*, (9–4), 2013.

75 Feedbackkultur im Unternehmen und Zufriedenheit von Mitarbeitern.

76 J. L. Fernández-Sánchez, M. García-García, J. García-Muñoz, and J. P. Gómez-Pérez. La ingeniería de sistemas y su aplicación a un vehículo aéreo no tripulado. *Dyna*, 87(4):456–466, 2012.

77 R. T. Fielding. *Architectural Styles andthe Design of Network-based Software Architectures*. University of California, Irvine PhD thesis, 2000

78 D. G. Firesmith, P. Capell, D. Falkenthal, C. B. Hammons, D. T. Latimer IV, and T. Merendino. *The Method Framework for Engineering System Architectures*. CRC Press, 2009.

79 K. Forsberg and H. Mooz. The relationship of systems engineering to the project cycle. In *Proceedings of the First Annual Conference of NCOSE*, 1991.

80 K. Forsberg, H. Mooz, and H. Cotterman. *Visualizing Project Management*. John Wiley & Sons, 2005.

81 E. Fourgeau, E. Gomez, H. Adli, C. Fernandes, and M. Hagege. System Engineering Workbench for Multi-Views Systems Methodology with 3DEXPERIENCE Platform. The Aircraft Radar Use Case. In M.-A. Cardin, S. H. Fong, D. Krob, P. C. Lui, and Y. H. Tan, editors, *Complex Systems Design & Management Asia*, pages 269–270. Cham, 2016. Springer International Publishing.

82 J. Fried and D. H. Hansson. *Rework*. Crown Business, 2010.

83 T. Fuchs, W. Gerritsen, K. Götz, J.G. Lamm, M. Pijorr, and T. Weilkiens. Die funktionale Architektur eines moderat-komplexen Systems standortübergreifend entwickeln. In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and S. Ackva, editors, *Tag des Systems Engineering 2019, München 6.-8. November 2019*, pages 33–42, Bremen, Germany, 2019. Gesellschaft für Systems Engineering.

84 E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

85 D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU/SEI-94-TR-21, Carnegie Mellon University, 1994.

86 A. Gawer. *Platforms, Markets and Innovation*. Edward Elgar, Cheltenham, U.K. and Northampton, M.A., 2009.

87 N. Gebhardt and D. Krause. *Methodische Entwicklung modularer Produktfamilien*. Springer.

88 W. Gerritsen. Personal communication. April 24th, 2018.

89 W. Gerritsen, J. G. Lamm, K. E. Neitzel, A. Scheithauer, C. Webel, and T. Weilkiens. Zielgerichtete Modellierung und stets aktuelle Views durch ein

präzises Viewpoint-Konzept im MBSE. In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and S. Ackva, editors, *Tag des Systems Engineering 2019, München 6.-8. November 2019*, pages 23–32, Bremen, Germany, 2019. Gesellschaft für Systems Engineering e.V.

90 GfSE PLM4MBSE Working Group. 10 theses about MBSE and PLM. https://www.gfse.de/Dokumente_Mitglieder/ag_ergebnisse/PLM4MBSE/PLM4MBSE_Position_paper_V_1_1.pdf, accessed November 2021.

91 D. Gianni, N. Lindman, J. Fuchs, and R. Suzic. Introducing the European space agency architectural framework for space-based systems of systems engineering. In O. Hammami, D. Krob, and J.-L. Voirin, editors, *Complex Systems Design & Management*, pages 335–346. Berlin Heidelberg, 2012, Springer.

92 J. L. Gibson, J. M. Ivancevich, J. H. Donnelly, and R. Konopaske. *Organizations: Behavior, Structure, Processes*. Mc-Graw-Hill, 14th edition, 2012.

93 D. Gray and T. Vander Wal. *The Connected Company*. O'Reilly, 2012.

94 D. Greefhorst and E. Proper. *Architecture Principles – The Cornerstones of Enterprise Architecture*. The Enterprise Engineering Series. Springer-Verlag, Berlin Heidelberg, 2011.

95 M. Grundel and J. Abulawi. SkiPo – A sketch and flow based model to develop mechanical systems. *INCOSE International Symposium*, 26(1):399–414, 2016.

96 M. Grundel, J. Abulawi, G. Moeser, T. Weilkiens, A. Scheithauer, S. Kleiner, C. Kramer, M. Neubert, S. Kümpel, and A. Albers. FAS4M – No more: "Please mind the gap!". In M. Maurer and S.-O. Schulze, editors, *Tag des Systems Engineering, Bremen, 12.-14. November 2014*, pages 65–74. München, Germany, 2014. Carl Hanser Verlag.

97 C. Guariniello, A. K. Raz, Z. Fang, and D. DeLaurentis. System-of-systems tools and techniques for the analysis of cyber-physical systems. *Systems Engineering*, 23(4):480–491, 2020.

98 C. Hampden-Turner and F. Trompenaars. Response to Geert Hofstede. *International Journal of Intercultural Relations*, 21(1):149–159, 1997.

99 C. Haskins, K. Forsberg, M. Krueger, D. Walden, and R. D. Hamelin. *INCOSE Systems Engineering Handbook v. 3.2*. INCOSE, January 2010.

100 M. Hassenzahl, A. Beu, and M. Burmester. Engineering joy. *IEEE Software*, 18(1):70–76, 2001.

101 L. Herrero. Viral Change™ The Alternative to slow, painful and unsuccessful management of change in organisations. In *Meetingminds, 2006*, 2008.

102 R. Hilliard. Lessons from the unity of architecting. In I. Jacobson and H. Lawson, editors, *Software Engineering in the Systems Context*, volume 7 of *Systems*, pages 225–250. London, 2015, College Publications.

103 D. K. Hitchins. *Systems Engineering*. John Wiley & Sons, 2007.

104 J. Holt and S. Perry. *SysML for Systems Engineering*, volume 10 of *IET Professional Applications of Computing Series*. The Institution of Engineering and Technology (IET), 3rd edition edition, 2018.
105 IEC 81346-1:2009. Industrial systems, installations and equipment and industrial products — Structuring principles and reference designations — Part 1: Basic rules.
106 ISO 15704:2019. Enterprise modelling and architecture — Requirements for enterprise-referencing architectures and methodologies.
107 ISO/IEC 26550:2015. Software and systems engineering — Reference model for product line engineering and management.
108 ISO/IEC 26580:2021. Software and systems engineering — Methods and tools for the feature-based approach to software and systems product line engineering.
109 ISO/IEC/IEEE 21841:2019. Systems and software engineering — Taxonomy of systems of systems.
110 ISO/IEC/IEEE 42020:2019. Software, systems and enterprise — Architecture processes.
111 ISO/IEC/IEEE 42030:2019. Software, systems and enterprise — Architecture evaluation framework.
112 ISO/IEC 7498-1:1984. Information processing systems — Open systems interconnection — Basic reference model.
113 ISO/IEC 25010:2011. Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE).
114 ISO/IEC/IEEE 42010:2011. Systems and software engineering — Architecture description.
115 ISO/IEC/IEEE 15288:2015. Systems and software engineering — System life cycle processes.
116 ISO/IEC/IEEE 24748-1:2018. Systems and software engineering — Life cycle management — Part 1: Guidelines for life cycle management.
117 ISO/IEC/IEEE 29148:2018. Systems and software engineering — Life cycle processes — Requirements engineering.
118 ISO 13849-1:2006. Safety of machinery — Safety-related parts of control systems — Part 1: General principles for design.
119 ISO Online Browsing Platform (OBP). http://www.iso.org/obp, accessed 3 August 2020.
120 IEEE Std 1471-2000. IEEE recommended practice for architectural description of software-intensive systems.
121 INCOSE. *Feature-based Systems and Software Product Line Engineering: A Primer*. INCOSE.
122 INCOSE Technical Operations. Systems engineering vision 2020. Version 2.03. Seattle, WA: International Council on Systems Engineering,

INCOSE-TP-2004-004-02. Available at http://www.incose.org/ProductsPubs?products/sevision2020.aspx, accessed September 2007.

123 INCOSE. Systems Engineering Vision 2025: The World in Motion. Available at https://www.incose.org/products-and-publications/se-vision-2025, accessed November 2021.

124 INCOSE Competency Working Group. INCOSE systems engineering competency framework – INCOSE-TP-2018-002-01.0. In *International Council on Systems Engineering (INCOSE)*, 2018.

125 INCOSE MBSE Challenge Team SE2. *Cookbook for MBSE with SysML*, 2011.

126 J. Isasa, P. Larsen, and F. Hansen. A holistic approach to energy-aware design of cyber-physical systems. *International Journal of Embedded Systems*, 9:283, 2017.

127 D. D. Jackson, P. Watzlawick, and J. B. Bavelas. *Pragmatics of Human Communication: A Study of Interactional Patterns, Pathologies and Paradoxes*. W. W. Norton, 2011.

128 I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1992.

129 I. Jacobson, I. Spence, and K. Bittner. Use-Case 2.0 - The Guide to Succeeding with Use Cases, 2011.

130 M. Jamshidi. *System of Systems Engineering: Innovations for the 21st Century*, volume 58. John Wiley & Sons, 2009.

131 D. Jemilo and D. Leffingwell. Safe 5 for Lean Enterprises. https://www.scaledagileframework.com, accessed November 2021.

132 H. D. Jørgensen, T. Liland, and S. Skogvold. Aligning TOGAF and NAF - Experiences from the Norwegian Armed Forces. In P. Johannesson, J. Krogstie, and A. L. Opdahl, editors, *The Practice of Enterprise Modeling*, volume 92 of *Lecture Notes in Business Information Processing*, pages 131–146. Berlin Heidelberg, 2011, Springer.

133 C. G. Jung. *Psychologische Typen*. Rascher Verlag, 1921.

134 C. G. Jung. *Psychological Types*, volume 20 of *Bollingen Series*. Pantheon Books, 1971.

135 K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

136 K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19:58–65, 2002.

137 R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon University, August 2000.

138 T. Kelley. *The Ten Faces of Innovation*. Profile Books, 2006.

139 M. Kennedy. Tate offers chance to experience night at the museum, with the help of four robots. *The Guardian (London / Manchester edition, United Kingdom)*, August 13th 3, 2014.

140 K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

141 S. K. Khaitan and J. D. McCalley. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal*, 9(2):350–365, 2015.

142 H. Kissinger. *Diplomacy*. Touchstone / Simon & Schuster, 1994.

143 D. A. Kolb. *Experiential Learning: Experience As The Source Of Learning and Development*. Prentice Hall, Englewood Cliffs, NJ, 1984.

144 A. Korff, J. G. Lamm, and T. Weilkiens. Werkzeuge für den Schmied funktionaler Architekturen. In M. Maurer and S.-O. Schulze, editors, *Tag des Systems Engineering, Hamburg, 9.-11. November 2011*, pages 3–12. München, Germany, 2011. Carl Hanser Verlag.

145 A. Kossiakoff and W. N. Sweet. *Systems Engineering — Principles and Practice.* John Wiley & Sons, Hoboken, New Jersey, 2003.

146 R. Krikhaar, W. Mosterman, N. Veerman, and C. Verhoef. Enabling system evolution through configuration management on the hardware/software boundary. *Systems Engineering*, 12(3):233–264, 2009.

147 P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

148 U. Könnemann, S. Stangl, M. Andres, and O. Bleisinger. Technical application of the system architecture framework (SAF). In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and S. Ackva, editors, *Tag des Systems Engineering 2019, München 6.-8. November 2019*, pages 13–22. Bremen, Germany, 2019. Gesellschaft für Systems Engineering e.V.

149 C. F. Kurtz and D. J Snowden. The new dynamics of strategy: Sense-making in a complex and complicated world. *IBM Systems Journal*, 42(3):462–483, 2003.

150 J. Lamm. *Exploration von Systemtests für die digitale Signalverarbeitung in Hörgeräten*. Shaker Verlag, Düren, Germany, 2019.

151 J. G. Lamm, A. Lohberg, and T. Weilkiens. Funktionale Architekturen in der Systementwicklung anwenden. In Maik Maurer and Sven-Olaf Schulze, editors, *Tag des Systems Engineering, Stuttgart, 6.-8. November 2013*, pages 283–292. München, Germany, 2013. Carl Hanser Verlag.

152 J. G. Lamm, A. Mettauer, G. Moeser, T. Weilkiens, and A. Albers. Storyboards in der Systementwicklung: eine neue Methode und ihr Zusammenspiel mit der FAS-Methode. In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and

S. Ackva, editors, *Tag des Systems Engineering, Paderborn, November 2017*, pages 105–114. München, Germany, 2017. Carl Hanser Verlag.

153 J. G. Lamm and T. Weilkiens. Funktionale Architekturen in SysML. In M. Maurer and S.-O. Schulze, editors, *Tag des Systems Engineering, München Freising, 10.-12. November 2010*, pages 109–118. München, Germany, 2010. Carl Hanser Verlag.

154 J. G. Lamm and T. Weilkiens. Method for deriving functional architectures from use cases. *Systems Engineering*, 17(2):225–236, 2014.

155 C. Larman and B. Vodde. *Practices for Scaling Lean and Agile Development*. Addison-Wesley / Pearson, 2010.

156 H. Lawson. *A Journey Through the Systems Landscape*, volume 1 of *Systems*. College Publications, 2010.

157 G. H. Lewes. *Problems of Life and Mind*. Number 2 in 1. Mifflin/Trübner, Houghton, M.I., 1875.

158 V.-C. Liang and C. J. J. Paredis. A port ontology for conceptual design of systems. *Journal of Computing and Information Science in Engineering*, 4:206–217, 2004.

159 J. Lockett and J. Powers. *Handbook of Human Systems Integration*, chapter Human Factors Engineering Methods and Tools, pages 463–496. John Wiley & Sons, 2003.

160 N. Luhmann. *A Sociological Theory of Law*. Routledge, 2013.

161 A. M. Madni. Expanding stakeholder participation in upfront system engineering through storytelling in virtual worlds. *Systems Engineering*, 18(1):16–27, 2015.

162 M. W. Maier. Architecting Principles for Systems-of-Systems. *Systems Engineering*, 1(4):267–284, 1998.

163 M. W. Maier. System and software architecture reconciliation. *Systems Engineering*, 9(2):146–159, 2006.

164 M. W. Maier and E. Rechtin. *The Art of Systems Architecting*. CRC Press, Boca Raton, F.L., 2nd edition, 2002.

165 J. N. Martin. *Systems Engineering Guidebook: A Process for Developing Systems and Products*. CRC Press, Inc, 1996.

166 J. N. Martin. Using the PICARD theory of systems to facilitate better systems thinking. *INCOSE INSIGHT*, 11(1):37–41, 2008.

167 MBSE Wiki Article – Model Based Document Generation. http://www.omgwiki.org/MBSE/doku.php?id=mbse:telescope#model_based_document_generation, accessed 19 June 2014 2013.

168 B. McCarthy. About Learning - Official Site of Bernice McCarthy's 4MAT System. http://www.aboutlearning.com, accessed 15 February 2015, 2015.

169 T. McDermott, D. DeLaurentis, P. Beling, M. Blackburn, and M. Bone. AI4SE and SE4AI: A Research Roadmap. *INSIGHT*, 23:8–14. https://doi.org/10.1002/inst.12278.

170 Megatrend Konnektivität. https://www.zukunftsinstitut.de/dossier/megatrend-konnektivitaet/, accessed 28 November 2020.

171 G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

172 MOD Architecture Framework - Detailed guidance. https://www.gov.uk/mod-architecture-framework, accessed 20 February 2015, December 2012.

173 G. Moeser, A. Albers, and S. Kümpel. Usage of free sketches in MBSE raising the applicability of model-based systems engineering for mechanical engineers. In *2015 IEEE International Symposium on Systems Engineering (ISSE)*, pages 50–55, 2015.

174 G. Moeser, C. Kramer, M. Grundel, M. Neubert, S. Kümpel, A. Scheithauer, S. Kleiner, and A. Albers. Fortschrittsbericht zur modellbasierten Unterstützung der Konstrukteurstätigkeit durch FAS4M. In S.-O. Schulze and C. Muggeo, editors, *Tag des Systems Engineering, Ulm, 11.-13. November 2015*, pages 69–78, München, Germany, 2015. Carl Hanser Verlag.

175 G. Moeser et al.. Model-based mechanical concept design: results of the FAS4M project. In S.-O. Schulze et al., editor, *Tag des Systems Engineering, Herzogenaurach, November 2016*. Carl Hanser Verlag, 2016.

176 J. P. Monat. Why customers buy — a look at how industrial customers make purchase decisions. *Marketing Research*, 28:20–24, 2009.

177 G. E. Moore. Cramming more components onto integrated circuits, 1965.

178 G. Muller. *Systems Architecting — A Business Perspective*. CRC Press, 2011.

179 G. Muller. The Role and Task of the System Architect. http://www.gaudisite.nl/info/RoleSystemArchitect.info.html, accessed 30 July 2014, 2013.

180 Federal Aviation Administration. FAA Systems Engineering Manual version 1.0.1, 2014.

181 NAVAIR Community of Practice. DoD Digital Engineering (DE) Initiatives. https://community.apan.org/wg/navair-set/navair-mbse-community-of-practice/w/resources/21397/dod-digital-engineering-de-initiatives/, November 2021.

182 Object Management Group (OMG): MDA Guide Version 1.0.1. OMG Document Number OMG/2003-06-01, 2003.

183 Object Management Group (OMG): OMG Systems Modeling Language (OMG SysML) Version 1.0. OMG Document Number formal/07-09-01, 2007.

184 Object Management Group (OMG): Requirements Interchange Format (ReqIF) Version 1.1.. OMG Document Number formal/2013-10-01, 2013.

185 Object Management Group (OMG): Semantics Of A Foundational Subset For Executable UML Models (FUML) Version 1.1, OMG Document Number formal/2013-08-06, 2013.
186 Object Management Group (OMG): Business Motivation Model (BMM) Version 1.2. OMG Document Number formal/2014-05-01, 2014.
187 Object Management Group (OMG): XML Metadata Interchange (XMI) Version 2.4.2.. OMG Document Number formal/2014-04-04, 2014.
188 Object Management Group (OMG): OMG Unified Modeling Language (OMG UML) Version 2.5.1. OMG Document Number formal/2017-12-06, 2017.
189 Object Management Group (OMG): OMG Systems Modeling Language (OMG SysML™) Version 1.6. OMG Document Number formal/19-11-01, 2019.
190 Object Management Group (OMG): Unified Architecture Framework® Profile (UAFP), Version 1.1. OMG Document Number formal/19-11-07, 2020.
191 Object Management Group (OMG): Unified Architecture Framework® (UAF®) Domain Metamodel, Version 1.1. OMG Document Number formal/19-11-05, 2020.
192 T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1978.
193 Open Services for Lifecycle Collaboration. https://www.open-services.net/, November 2021.
194 B. W. Oppenheim. *Lean for Systems Engineering with Lean Enablers for Systems Engineering*. John Wiley & Sons, 2011.
195 T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.
196 Oxford Dictionary on Lexico.com. https://www.lexico.com, accessed 3 August 2020.
197 G. Pahl, W. Beitz, J. Feldhusen, and K.-H. Grote. *Engineering Design*. Springer, 2007.
198 D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
199 G. Patzak. *Systemtechnik, Planung Komplexer Innovativer Systeme*. Springer, 1982.
200 N. Plum. TRAK Enterprise Architecture Framework. http://trak.sourceforge.net, accessed 30 July 2014, 2010.
201 K. Pohl. *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, 2010.
202 K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1st edition, 2005.
203 V. Pollio. De Architectura Liber Primus. In *De Architectura Libri Decem*, Chapter 3. first century bce.

204 T. Rabbets, M. Wilkinson, H. Sillitto, K. Galvin, P. Handisides, G. Bleakley, B. Madahar, and K. Akroyd-Wallis. Introduction to System Architecting – An INCOSE UK Guide.

205 F. Rahimi and A. Ipakchi. Demand Response as a Market Resource Under the Smart Grid Paradigm. *IEEE Transactions on Smart Grid*, 1(1):82–88, 2010.

206 E. Rauch, D. T. Matt, C. A. Brown, W. Towner, A. Vickery, and S. Santiteerakul. Transfer of Industry 4.0 to Small and Medium Sized Enterprises. *Transdisciplinary Engineering Methods for Social Innovation of Industry*, 4:63–71, 2018.

207 D. J. Richardson and L. A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 244–253. San Diego, CA, March 1981. IEEE.

208 M. Richter and M. D. Flückiger. *Usability Engineering Kompakt*. Springer Vieweg, 2013.

209 S. Robertson and J. Robertson. *Mastering the Requirements Process: Getting Requirements Right*. Addison-Wesley Professional, 3rd edition, 2012.

210 P. Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7, 1986.

211 M. Roscia, M. Longo, and G. C. Lazaroiu. Smart city by multi-agent systems. In *2013 International Conference on Renewable Energy Research and Applications (ICRERA)*, pages 371–376, 2013.

212 W. W. Royce. Managing the development of large software systems. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, IEEE Computer Society Press, Washington, 1987, pages 328–338.

213 N. Rozanski and E. Woods. *Software Systems Architecture*. Addison Wesley / Pearson, 2012.

214 D. Scheithauer and K. Forsberg. V-Model views. In *Proceedings of the 23nd Annual International Symposium of the International Council on Systems Engineering*, 2013.

215 G. Schuh. *Produktkomplexität Managen*. Carl Hanser Verlag München, 2005.

216 K. Schwaber. Scrum development process. In J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, editors, *Business Object Design and Implementation*, pages 117–134. Springer.

217 K. Schwaber et al. The Nexus™ Guide, January 2021. https://www.scrum.org/resources/nexus-guide, accessed October 2021.

218 M. L. Scott and W. Ambler. Disciplined agile delivery: an introduction.

219 SEBoK Authors. *Digital Engineering*. Hoboken, N.J., The Trustees of the Stevens Institute of Technology. BKCASE is managed and maintained by the Stevens Institute of Technology Systems Engineering Research Center, the International Council on Systems Engineering, and the Institute of Electrical and Electronics Engineers Systems Council.

220 SEBoK Authors. *Functional Architecture (Glossary)*. Hoboken, N.J., The Trustees of the Stevens Institute of Technology. BKCASE is managed and maintained by the Stevens Institute of Technology Systems Engineering Research Center, the International Council on Systems Engineering, and the Institute of Electrical and Electronics Engineers Systems Council.

221 L. J. Seiwert. *Slow Down to Speed Up*. Campus Verlag, 2008.

222 S. A. Selberg and M. A. Austin. Toward an Evolutionary System of Systems Architecture. *INCOSE International Symposium*, 18(1):1065–1078, 2008.

223 P. M. Senge. *The Fifth Discipline: The Art and Practice of the Learning Organization*, Doubleday/Currency, 1st edition, 1990.

224 C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27: 1–3, 1948.

225 Sierpinski triangle evolution. https://commons.wikimedia.org/wiki/File:Sierpinski_triangle_evolution.svg, accessed 21 February 2021.

226 H. Sillitto. *Architecting Systems*, volume 6 of *Systems*. College Publications, London, 2014.

227 H. Sillitto, J. Martin, D. McKinney, R. Griego, D. Dori, D. Krob, P. Godfrey, E. Arnold, and S. Jackson. INCOSE-TP-2020-002-06, Systems Engineering and System Definitions, 2019.

228 H. A. Simon. A Behavioral Model of Rational Choice. *The Quarterly Journal of Economics*, 69(1):99–118, 1955.

229 D. C. Skinner. *Decision Analysis*. Probabilistic Publishing, 2001.

230 Smart Systems - Vernetzte, intelligente Systeme für moderne Anwendungen. https://www.wirtschaft-digital-bw.de/themen/thema-des-monats/smart-systems-vernetzte-intelligente-systeme-fuer-moderne-anwendungen/, accessed 16 November 2020.

231 H. Song, D. B. Rawat, S. Jeschke, and C. Brecher. *Cyber-Physical Systems: Foundations, Principles and Applications*. Academic Press, Inc., Orlando, F.L., 1st edition, 2016.

232 D. E. Spielberg. *Methodik zur Konzeptfindung basierend auf technischen Kompetenzen*. Doctoral thesis, Shaker-Verlag, RWTH Aachen, 2002.

233 H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

234 J. Stapleton. *DSDM Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley, Harlow, 1997.

235 M. Stickdorn and J. Schneider. *This is Service Design Thinking*. BIS Publishers, Amsterdam, The Netherlands, 2013.

236 N. P. Suh. *The Principles of Design*. Oxford University Press, New York, 1990.

237 J. Sutherland et al. The Scrum At Scale® Guide", https://www.scrumatscale.com/scrum-at-scale-guide/, accessed January 2021.

238 J. Sutherland and K. Schwaber. The scrum guide, https://scrumguides.org, accessed November 2020.

239 SysML v2 API and Services RFP.
240 Systems Modeling Language (SysML®) v2 Request for Proposal (RFP).
241 H. Takeuchi and I. Nonaka. The new new product development game. *Harvard Business Review*, 137–146, 1986.
242 C. Talcott. Cyber-Physical Systems and Events. In M. Wirsing, J.-P. Banâtre, M. Hölzl, and A. Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms: Challenges and Visions*, pages 101–115. Berlin, Heidelberg, Springer, 2008.
243 The DSDM Agile Project Framework.
244 K. Beck, M. Beedle, A. van Bennekum et al. The Manifesto for Agile Software Development. https://agilemanifesto.org, accessed November 2021.
245 The Open Group. The TOGAF® Standard, Version 9.2, 2018. https://www.opengroup.org/togaf, accessed November 2021.
246 D. Thomas. Agile is Dead (Long Live Agility). https://pragdave.me/blog/2014/03/04/time-to-kill-agile.html, accessed November 2021.
247 C. Thronesbery, A. Molin, and D. L. Schreckenghost. A storyboard tool to assist concept of operations development. In *2007 IEEE Aerospace Conference*, pages 1–8, 2007.
248 F. Trompenaars and C. Hampden-Turner. *Riding The Waves Of Culture*. Nicholas Brealey Publishing, 2012.
249 K. M. Tsui, M. Desai, H. A. Yanco, and C. Uhlik. Exploring use cases for telepresence robots. In *2011 6th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 11–18, 2011.
250 B. W. Tuckman. Developmental sequence in small groups. *Psychological Bulletin*, 63(6):384–399, 1965.
251 B. W. Tuckman and M. A. C. Jensen. Stages of small-group development revisited. *Group & Organization Studies*, 2(4):419–427, 1977.
252 D. G. Ullman. *Making Robust Decisions*. Trafford Publishing, 2006.
253 K. Ulrich. The role of product architecture in the manufacturing firm. *Research Policy*, 24:419–440, 1995.
254 UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification Version 1.2. OMG Document Number formal/2010-06-01, 2010.
255 U.S. Chief Information Officer. DoDAF - DOD Architecture Framework Version 2.02. http://dodcio.defense.gov/TodayinCIO/DoDArchitectureFramework.aspx, accessed 20.02.2015, 2010.
256 R. Valerdi, E. Axelband, T. Baehren, B. Boehm, D. Dorenbos, S. Jackson, A. Madni, G. Nadler, P. Robitaille, and S. Settles. A research agenda for systems of systems architecting. *International Journal of System of Systems Engineering*, 1(1/2):171–188, 2008.

257 VDI 2206:2004. Entwicklungsmethodik für mechatronische Systeme / Design methodology for mechatronic systems, 2004.
258 Las Vegas. The Mob Museum. News page. https://themobmuseum.org/media-room/in-the-news/, visited January 1st, 2021.
259 V-Modell XT. https://www.cio.bund.de/Web/DE/Architekturen-und-Standards/V-Modell-XT/vmodell_xt_node.html, accessed 26 June 2021.
260 B. Vodde et al. Overview - Large Scale Scrum (Less). https://less.works, accessed November 2021.
261 A. Vollerthun. Design-to-market integrating conceptual design and marketing. *Systems Engineering*, 5(4):315–326, 2002.
262 L. von Bertalanffy. General System Theory, 1968.
263 F. S. von Thun. *Miteinander reden 1-4: Störungen und Klärungen. Stile, Werte und Persönlichkeitsentwicklung. Das "Innere Team" und situationsgerechte Kommunikation. Fragen und Antworten*. Miteinander reden. Rowohlt Taschenbuch Verlag, 2014.
264 B. Waber, J. Magnolfi, and G. Lindsay. Der Wert der Gestaltung. *Harvard Business Manager*, 2/2015, 2015.
265 D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin, and T. M. Shortell. *INCOSE Systems Engineering Handbook*. John Wiley & Sons, 4th edition, 2015.
266 M. Walker. What is architecture? In S.-O. Schulze, C. Tschirner, R. Kaffenberger, and S. Ackva, editors, *Tag des Systems Engineering 2019, München 6.-8. November 2019*, Bremen, Germany, 2019. Gesellschaft für Systems Engineering.
267 T. Weilkiens. *Systems Engineering with SysML/UML*. Morgan Kaufmann OMG Press, 2008.
268 T. Weilkiens. A system architecture is what system architects create (blog entry). http://model-based-systems-engineering.com/2014/03/10/a-system-architecture-is-what-system-architects-create/, accessed January 16th, 2022, March 2014.
269 T. Weilkiens. *Systems Engineering mit SysML/UML*. dpunkt-Verlag, 2014.
270 T. Weilkiens. *Variant Modeling with SysML*. MBSE4U, 2015.
271 T. Weilkiens. *SYSMOD - The Systems Modeling Toolbox*. MBSE4U, 3rd edition, 2020.
272 E. Wenger. *Communities of Practice*. Cambridge University Press, 1998.
273 C. Duhigg. What Google Learned From Its Quest to Build the Perfect Team. *The New York Times Magazine*, 25 Feb 2016.
274 What is your definition of software architecture? http://www.sei.cmu.edu/architecture/start/glossary/definition-form.cfm, accessed 23 October 2020.
275 M. K. Wilkinson, C. E. Dickerson, and S. Ji. Concepts of architecture, structure and system, 2018.

276 G. Wohland and M. Wiemeyer. *Denkwerkzeuge der Höchstleister*. Murmann, 1st edition, 2007. 3rd edition, UNIBUCH Verlag, 2012.

277 A. A. Yassine and L. A. Wissmann. The implications of product architecture on the firm. *Systems Engineering*, 10(2):118–137, 2007.

278 L. Zhang. Modeling large scale complex cyber physical control systems based on system of systems engineering approach. In *2014 20th International Conference on Automation and Computing*, pages 55–60. IEEE, 2014.

279 C. Zingel, A. Albers, S. Matthiesen, and M. Maletz. Experiences and advancements from one year of explorative application of an integrated model-based development technique using C&C^2-A in SysML. *IAENG International Journal of Computer Science*, 34–39(2):165–181, 2012.

Index

«I/O» 251
«activity» 356
«adjunct» 357
«constraint» 361
«continuousUseCase» 364
«deriveReqt» 79
«domainBlock» 110, 355
«full» 358
«functionalBlock» 246
«functionalGroup» 246
«proxy» 359
«requirement» 373
«systemUseCase» 364
«userInterface» 250, 360
«user» 364
«variantConfiguration» 275
«variant» 273
«variation» 273
«verify» 362
4MAT System 340
7 ± 2 192

a

abstract syntax 48, 90, 352
abstraction 46, 96, 131, 188, 269
abstraction level 76, 93
accept event action 246, 369
acquirer 161
act, *see* activity, diagram
action 364, 366
  opaque, *see* opaque action
  type 367
activity 245, 364, 366
  diagram 109, 130, 245, 257, 363, 366
  final node 369
  parameter 367
  partition 251, 367
  tree 109, 243, 244, 356
actor 147, 249, 364
AD element, *see* architecture description, element
adapter 103
adjunct property 245, 357
ADL, *see* architecture description language
ADM, *see* Architecture Development Method
agile 117, 213, 221, 222
  corporate culture 213
  framework 213, 223
  guild 230
  mindset 213
  principle 213, 214, 221, 223, 226
  value 213, 214
Agile Business Consortium 219
AI, *see* artificial intelligence
Allen Curve 333
allocation 255

Model-Based System Architecture, Second Edition. Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker.
© 2022 John Wiley & Sons, Inc. Published 2022 by John Wiley & Sons, Inc.

allocation matrix 150, 157
alternative 371
anti-pattern 224
Apollo program 214
approval 204
ArchiMate 292
architect, *see* system architect
architectural style 32, 310
architecture 33, 34, 38, 43, 68, 76, 350
  assessment team 309
  aspect, *see* perspective
  assessment 307
  base, *see* base architecture
  concept selection 37
  context interaction, *see* context interaction
  decision, *see* architecture decision
  definition 31–37
  description, *see* architecture description
  design principle, *see* design principle
  discipline specific, *see* discipline specific architecture
  discussion of definitions 31
  element 36
  evolution principle, *see* evolution principle
  framework, *see* architecture framework
  functional, *see* functional architecture
  landscape 41
  layered, *see* layered architecture
  logical, *see* logical architecture
  model (ISO 42010), *see* visualization
  model kind (ISO 42010), *see* visualization kind
  modular 202
  organization principle, *see* organization principle
  pattern 75
  perspective, *see* perspective
  physical, *see* physical architecture
  principle 75, 244, 250
  product, *see* product architecture
  rationale, *see* architecture rationale
  references to definitions 31
  stakeholder, *see* architecture stakeholder
  system, *see* system architecture
  system design, *see* system design
  system element interaction, *see* system element interaction
  view, *see* architecture view
  viewpoint, *see* architecture viewpoint
architecture decision 33, 36–39, 57, 60, 61, 67–70, 74, 82, 201, 242, 310
architecture description 33, 34, 57, 84, 96, 119, 123, 133, 148, 156, 164, 166, 167, 187, 199, 203
  absence of 58
  definition 60–69
  forms and templates 71–74
  in BRS 71
  in ConOps 71
  in OpsCon 71
  in StRS 71
  in SyRS 71
  model-based vision 69–71
  project 73
  purpose of 57
architecture description element 60, 61, 68
architecture description language 58, 62–64, 74
Architecture Development Method 286
architecture evaluation 307
architecture evaluation framwork 307

architecture framework 62, 63, 65, 120, 279, 280, 296
  overview of 285
architecture governance 181, 183, 186, 199, 201
architecture rationale 57, 60, 61, 68–69, 74
architecture stakeholder 59, 67, 119, 156, 157, 161, 162, 165, 210
Architecture Tradeoff Analysis Method 307
architecture view 16, 60–67, 73, 119, 120, 124, 126, 127, 129, 137, 142, 155–157, 159
architecture viewpoint 60–66, 73, 120, 159
argument 69
arranged, *see* organization principle
artificial intelligence 5, 13, 302, 347
aspect, *see* perspective
assembly, *see* system element
assessment 250
association 355, 364
assumption 69
asynchronous message 372
ATAM, *see* Architecture Tradeoff Analysis Method
authority 182
automation 251
availability 308
axiomatic design 78

b

base architecture 37, 39, 41, 57, 73, 82, 126, 279, 309
baseline 159, 166, 167
bdd, *see* block definition diagram
BDUF, *see* big design up front
behavior 130, 175, 366
  diagram 349, 363
  flow-oriented, *see* flow-oriented behavior
behavioral port 249, 359
benefit 59
big design up front 216, 227
bill of material 358
binding connector 359
block 246, 354
block definition diagram 123, 127, 244, 246, 353, 354
block diagram 353
blog 336
BMM, *see* Business Motivation Model
BOM, *see* bill of material
brand management 178
BRS, *see* business requirements specification
budget 172, 306
building block, *see* system element
business driver 309
business goal 281, 282, 309
Business Motivation Model 91
business requirements specification 71
business system context 309

c

C4ISR 289
CAD, *see* computer-aided design
call behavior action 109, 245, 368
call operation action 246
call-out notation 362
capacity 174
card technique 258
Carnegie Mellon University 307
category partition method 165
CDF, *see* concurrent design facility
change process 313
chat 336
checklist 75

CIM, *see* Computational Independent Model
classification tree 165
cohesion 85, 243
collaborative planning methodology 288
color-blinded 93
combined fragment 372
communication 14, 49, 57, 58, 62, 81, 162, 164, 175, 183, 188, 201, 327, 328, 344, 387
communication protocol 371
compatibility 166, 167
component system, *see* constituent system
composite property 358
composite relationship 248, 354
Computational Independent Model 48
computer-aided design 145
concept model 110
concept of operations 57, 71
concern 57, 58, 61–63, 66, 68, 73, 119, 156, 161
concrete syntax 48, 91, 352
concurrent design facility 334
configuration 269
  item 166, 167
  management 159, 166, 167, 207
  manager 167
conflicts 182
connector 124, 358
connotation 335
ConOps, *see* concept of operations
consistency 16, 156
consolidated reference model 288
constituent, *see* system element
constituent system 23, 282–284
constraint 69, 144, 260, 354, 361
constraint block 361
constraint property 361
constraint requirement 85
constructionist perspective 22
consumer goods 10
context, *see* system context
context interaction 19, 36, 70
context interface layer 111
control flow 109, 244, 366
control node 366
ControlValues library 351
core activity 238
core element 274
  definition 270
cost 14, 157
coupling 85, 131
CPS, *see* cyber-physical system
CPSoS, *see* cyber-physical system of systems
CRYSTAL 47
CS, *see* constituent system
cultural gap 235
culture 92, 344
customized product 269
cyber-physical system 26–27, 302
cyber-physical system of systems 27–30
cyber-security 12, 302–304

d

database model 111
decision
  in architecture description, *see* architecture decision
decision node 366
decompose requirements 372
definition 87
denotation 335
Department of Defense Architecture Framework 289, 290
dependency 176
deployment 120, 129, 143, 144
derive relationship 79
  extended 79
design 43

design definition process 152, 207
design principle 38, 39, 70
development roadmap 175
diagram 352
  header 353
  layout 92
digital engineering 45
digital thread 45
digital twin 45
discipline 43, 44
discipline specific architecture 44
discipline specific design 44
discriminator 271
disposability 308
distribution 177
diversity 342
document-based 48, 155, 204
DoDAF, *see* Department of Defense Architecture Framework
DoDAF Meta Model 289
domain 40
domain knowledge 40, 69, 110, 121, 127, 133–138, 157, 158, 355
domain knowledge model 128, 158
domain model 110, 355
drawing 91
DSDM, *see* dynamic systems development method
dynamic systems development method 219, 221
  agile project framework 219
  consortium 219

e

e-mail 333–335
EA, *see* enterprise architecture
effort 171
effort estimation 172
element, *see* system element
emergence 18, 19, 30, 283
emergent
  behavior 30, 283, 284
  property 30, 283
emotions 180
enabling system 162, 316
enterprise 280–282
enterprise architecture 280, 282, 285
entity, *see* system element
entry behavior 370
environment, *see* system context
environmental effect 105, 144
equivalence class 165
ESA, *see* European Space Agency
ESA-AF, *see* European Space Agency Architectural Framework
European Space Agency 293, 334
European Space Agency Architectural Framework 294
event-oriented behavior 363
everything as a service 28
evidence 69
evolution, *see* system evolution
evolution principle 19, 23, 36, 37, 70
evolutionary process 37
example scenario 371
example system 5
execution order 366
exit behavior 370
explanation 57
extension mechanism 374
extension relationship 375
external interaction, *see* context interaction
external interface 104, 162
external system 364
extreme programming 215

f

family of systems 282, 283
FAS
  engine 251
  heuristics 242

FAS (*contd.*)
method 121, 124, 197, 227, 233–254, 256, 322
profile 246
workshop 227, 258
FAS4M 145
fault-tolerance 261
FEAF, *see* Federal Enterprise Architecture Framework
FEAPO, *see* Federation of Enterprise Architecture Professional Organizations
feasibility 171, 175, 207
feature 234, 271, 309
team 190, 203, 205
Feature Oriented Domain Analysis 274, 276
feature team 170
feature tree 274, 276
Feature-Oriented Reuse Method 277
Federal Enterprise Architecture Framework 288, 289
Federation of Enterprise Architecture Professional Organizations 280
financial constraint 309
flipchart 91, 259, 338
flow 157, 168, 233
flow property 249, 358
flow-oriented behavior 363
FODA, *see* Feature Oriented Domain Analysis
fork node 366, 369
FORM, *see* Feature-Oriented Reuse Method
fUML, *see* Semantics Of A Foundational Subset For Executable UML Models
function 121, 175, 234, 236
function structure 246
function tree, *see* activity tree
function-mean tree 78
functional architecture 37, 41, 121, 145, 151, 233, 234, 236, 244, 249, 254, 309
concept selection 39
connection 38
definition 37–39
element 38
Functional Architecture for Systems, *see* FAS
functional connection 238, 241
functional decomposition 102, 236
functional element 121, 236, 241
definition 238
functional group 240, 244
definition 238
top level 253
functional interaction point 238
definition 238
functional interface 238, 241
functional parameter 238
functional safety 235
functional-to-physical mapping 149, 151, 235, 255

g

generalization 274, 354
geometry 145
Google Aristotle 342, 343
graphical pattern 92
graphical representation 90, 92

h

happy engineering 174
heuristic 95, 242
human actor 105
human factors 180, 304

i

I/O activity 102, 238
I/O activity partition 110

I/O group 253
ibd, *see* internal block diagram
idea 33, 57
ideal system 89
implementation 168
implementation knowledge 43
implicit knowledge 75
imposed architecture, *see* base architecture
imposed solution, *see* base architecture
include relationship 90, 365
INCOSE, *see* International Council on Systems Engineering
INCOSE Systems Engineering Competency Framework 190
industry project 256
information flow 124
information hiding 134
information radiator 337
initial state 370
input pin 367
inside-out view 106
instant messaging 336
integral architecture 150
integrated, *see* organization principle
integration test 164
interaction 371
  context, *see* context interaction
  external, *see* context interaction
  internal, *see* system element interaction
  system element, *see* system element interaction
interaction point 358
interaction principle 70
interest 58
interface 121, 141, 157, 162, 168–170, 178, 179, 254
interface block 360
interface control document 126
internal block diagram 125, 129, 138, 147, 148, 157, 249, 353, 357, 361
internal interaction, *see* system element interaction
internal transition 370
International Council on Systems Engineering 45, 71, 72, 218, 225, 324, 349
  lean systems engineering working group 218
ISO 80000 library 111, 351
item flow 358
iterative-incremental approach 213–215, 223, 226
  time box 215

j

J2EE™, *see* Java2 Enterprise Edition
Jacobson, Ivar 117
Java2 Enterprise Edition 48
justification 19

k

keep it simple 97, 156
knowledge
  domain, *see* domain knowledge
  implementation, *see* implementation knowledge
  product, *see* product knowledge

l

large scale scrum 229
layer 153, 301
  definition 131
layered architecture 39, 139
leadership 324
leaf entity, *see* system element
lean 217
  kaizen 218
  kanban 218

lean (*contd.*)
management 217
muda 218
production 217
learning organization 214, 225, 226
Learning Style Inventory 340
LeSS, *see* large scale scrum
lifeline 372
lifetime of technologies 234
logical architecture 37, 39–41, 126, 309
definition 255
element 40
logistics 179
loop 371
LSI, *see* Learning Style Inventory

m

maintainability 10, 173
maintenance 134, 161
management 181–183
manifesto for agile software development 221–223
principles 222
manufacturability 179
mapping 46
market needs 180
marketing 161, 178, 180, 181
matrix representation 247, 373
matrix view 246
maxVariants 273
MBE, *see* model-based engineering
MBPLE, *see* model-based product line engineering
MBSA, *see* model-based system architecture
MBSE, *see* model-based systems engineering
MBSE cookbook 93
MDA, *see* Model Driven Architecture
megatrend
connectivity 25
informatization 27
message 371
message-oriented behavior 363
meta-model 285, 288, 290, 291, 294
metaclass 374
methodology 291
methodology adoption process 50
Microsoft.NET™ 48
Ministry of Defense Architecture Framework 290
minVariants 273
MODAF, *see* Ministry of Defense Architecture Framework
MODAF Meta Model 290
model 90, 352, 362
definition 46
model (ISO 42010), *see* visualization
Model Driven Architecture 48
model execution environment 368
model governance 51, 52, 54, 55
model kind (ISO 42010), *see* visualization kind
Model Lifecycle Management 55
model organization 362
model transformation 49
model-based
definition 48
model-based engineering 45
model-based systems architecting
definition 45
model-driven 48
model-based document generation 71, 205
model-based product line engineering 3, 269
model-based system architecture 1, 46
model-based systems engineering 45, 227
modeling 81
modeling tool 157, 158, 251

modifiability 200, 308
modular architecture 150

n

NAF, *see* NATO Architecture Framework
NASA, *see* National Aeronautics and Space Administration
National Aeronautics and Space Administration 214, 281
NATO, *see* North Atlantic Treaty Organization
NATO Architecture Framework 291
nested package 362
nested relationship 362
non-functional requirement 301
nonhuman actor 105
North Atlantic Treaty Organization 291

o

object flow 109, 244, 254, 366
Object Management Group 295, 349
objective, *see* purpose
OMG, *see* Object Management Group
OMG SysML, *see* Systems Modeling Language
OMG Unified Architecture Framework 295, 296
opaque action 246, 368
opaque behavior 368
Open Services for Lifecycle Collaboration 49
open source 251
operation 246, 247, 354, 372
operational concept 57, 71
operational context 105, 147
opportunity 69
OpsCon, *see* operational concept
organization 313
organization principle 18, 19, 36, 37, 70
organization(al) 162, 169, 172, 174–176, 178, 179, 182, 190–193, 318, 319
  interface 162, 319, 320
  structure 314, 317
organizational gap 235
organized, *see* organization principle
orthogonal aspects 93
Orthogonal Variability Model 270, 277
OSI reference model 132
OSLC, *see* Open Services for Lifecycle Collaboration
output pin 367
outside-in view 105
over-modeling 51, 120, 122
OVM, *see* Orthogonal Variability Model
owned property 354

p

package 362
package diagram 353, 362
package structure 93
par, *see* parametric diagram
parallel message exchange 371
parameter 367
parametric diagram 144, 353, 361
part, *see* system element
  in SysML 354, 358
pattern 75, 96
pattern language 75
PBS, *see* product breakdown structure
performance 308
performance indicator 307
personality types 338
perspective 63, 65, 120, 126, 155–157, 285, 301
  system deployment 120, 142
  behavioral 120, 130
  functional 120, 121
  geometry 145

perspective (*contd.*)
layered 120
mapping 121
physical 120, 125
topology 146
physical architecture 37, 39, 41
base, *see* base architecture
concept selection 39
definition 37–39, 255
element 38–40
layered, *see* layered architecture
logical, *see* logical architecture
product, *see* product architecture
taxonomy 39
physical effect 40
PIM, *see* Platform Independent Model
pkg, *see* package diagram
planning 12, 171, 176
Platform Independent Model 48
Platform Specific Model 49
PLE, *see* product line engineering
point of view
in architecture description, *see* architecture viewpoint
port 124, 141, 358
conjugated 360
full 358
proxy 141, 249, 358
potentially shippable product 224
potentially shippable product increment 224
pragmatic 46
PrimitiveValueTypes library 351
principle 32, 37, 69, 75, 95, 96
principle solution 39, 41
process 75, 200
systems architecting 199
producibility 308
product architecture 37, 39, 41, 78, 126, 139, 145, 151, 309
definition 255
element 40
product breaddown structure 92
product knowledge 41
product line 269
product line engineering 269
product tree 355
production 13, 161, 162, 177
profile 285, 375
profile application 362
profile application relationship 375
project 174
project leader 173
project manager 171, 172
Project Mercury 214
Friendship 7, 215
property 357
provided operation 358
provided structure 358
PSM, *see* Platform Specific Model
psychological safety 342–344
feedback culture 344
psychological types 338, 339
purpose 18, 19, 22, 59, 73

q

quality 259
quality attribute 308
quality attributes utility tree 310
quality criteria 10
quality of service 261
quality requirement 179, 180, 259
QUDV model library 351
quick win 182

r

RAD, *see* rapid application development
rapid application development 219

rationale
in architecture description, *see* architecture rationale
reading direction 92
reasoning 69, 70
receive signal 369
recruiting 192
reduction 46
redundancy 97
reference architecture 279
reference property 358
refine relationship 372
region 370
regressive verification 165
regulatory body 161
reliability 10, 308
repository
definition 47
req, *see* requirements diagram
requested operation 358
requested structure 358
requirement 11, 57, 60–63, 66–68, 76, 103, 122, 161, 165, 174, 239, 287, 372
constraint 102
discipline specific 78
functional 101
non-functional 101, 150
quality 101
specification 101
requirement identifier 372
requirement table 373
requirements analysis 99
requirements diagram 349, 372
requirements documentation 103
requirements elicitation 103
requirements engineer 161, 174, 235, 239, 258, 319
requirements engineering 103, 161, 164
requirements management 103
requirements structure 103
REQUIRES 271, 274
resources 97, 175–177
REST/HTTP 49
retirement 162
return of investment 14, 175, 181, 183
review 16, 204
risk 69, 175, 311
risk management 12, 171, 174, 305
roadmap 175
role 163, 185, 314, 317, 318, 324
description 185
RSS feed 336
run-time 87

S

SAF, *see* System Architecture Framework
SAFe, *see* scaled agile framework
safety 302, 304
safety assessment 236
SAMS method, *see* Storyboard Activity Modeling for Systems method
satisfy relationship 372
scaled agile framework 229
scaling framework 229
scenario 371
Scrum 220, 221, 226, 228
framework 221
guide 221
master 220
product owner 220
sprint 220
sprint retrospective 220
team 220
sd, *see* sequence diagram
security 12, 302, 304, 308
self-organized team 216, 225
semantic 352

Semantics Of A Foundational Subset For Executable UML Models 368
send signal 369
send signal action 246, 369
sensitivity point 311
sequence diagram 130, 363, 371
service-oriented 28
SESAR 292
  air traffic management 292
Shannon–Weaver model 329
shared aggregation 356
shared property 354
short message service 337
sierpinski triangle 24
signal 372
signal flow 124
simplify. simplify. simplify. 97
simulation 16, 368
simulation result 69
single source of truth 16, 157, 190, 321
sketch 91, 259
skill 188
SMAP, *see* methodology adoption process
SMS, *see* short message service
soft skills 327
software architect 168
software engineering 307
SoI, *see* system-of-interest
solution 43, 57
solution hypothesis 57
solution principle 39, 40
solution space 84
solution-free 82
solution-free requirement 76
SoS, *see* system of systems
SoSE, *see* system of systems engineering
space race 214
specialization 354
specification 49
spiral model 216–218
  risk-driven approach 216
stable part 89
stakeholder 16, 48, 51, 53, 54, 57, 58, 60, 61, 73, 97–99, 119, 129, 156, 157, 161, 162, 168, 174, 175, 192, 207, 210, 230, 314, 319–321, 323, 341, 348, 364
  architecture, *see* architecture stakeholder
stakeholder need 281
stakeholder process 58
stakeholder requirements specification 71, 76
standard profile 351
state 369
state machine 368, 369
state machine diagram 130, 363, 369
stereotype 285, 296, 375
sticky figure 364
stm, *see* state machine diagram
storyboard 112
Storyboard Activity Modeling for Systems method 112
strategy 10
StRS, *see* stakeholder requirements specification
structure 175
structure diagram 349, 353
sub-states 370
subsystem, *see* system element
subsystem development 12
supplier 178
support 162
synchronous message 372
SyRS, *see* business requirements specification
SysML 121, 129, 130, 139–141, 145, 150, 157, 159
  SysML 1.0 349

SysML model 93
SysML v2 350, 376
SYSMOD 76, 99, 163
  intensity model 49
  zigzag pattern, *see* zigzag pattern
SYSMOD profile 375
system 33, 34, 60, 70, 104
  architecture, *see* system architecture
  behavior, *see* system characteristic
  boundary 20
  characteristic, *see* system characteristic
  context, *see* system context
  context interaction, *see* context interaction
  definition 17–23
  design, *see* system design
  element, *see* system element
  element interaction, *see* system element interaction
  engineered 18
  environment, *see* system context
  evolution, *see* system evolution
  impact 58
  maintenance 57
  meaning, *see* system characteristic
  networked 26
  organization 36
  property, *see* system characteristic
  realization 57
  self-contained 27
  smart 27
  success 59
system actor 105
system architect 32, 58, 62, 67, 123, 153, 207, 235, 239, 258
  definition 186
system architecture 33, 34, 37, 42, 122
  definition 43
  interest in 59
System Architecture Framework 296, 297
system architecture team 190, 314, 320
system block 236
system boundary 104, 146, 147, 303
system characteristic 19, 21, 22, 59
system context 19, 20, 36, 60, 61, 70, 73, 104, 144, 146, 147, 165, 249, 258, 303
system context diagram 104, 149
system design 37, 43, 125, 126, 150–152, 207
system element 18–21, 36, 60, 61, 70, 121, 125, 143, 168, 170, 186
system element interaction 19, 36, 70, 121
system evolution 19, 23, 36, 57, 70
system function 106
system integration 169, 171
system model
  definition 46
system model structure 93
system of systems 23–26, 282–284
  architecture 280
system of systems engineering 280
system operational concept, *see* operational concept
system requirements specification 71, 77
system test 165
system-of-interest 20, 25, 57, 58, 60, 61, 73, 119, 146, 152
systems engineering 225
systems engineering vee, *see* V-Model
Systems Modeling Language 349
systems modeling toolbox, *see* SYSMOD

t

t-shaped 193, 195
table representation 373
tagged value 375
TDD, *see* test-driven development
technical decision 82
technology-independent 234
telephone conference 337
temporal cohesion 106
terminology 157
test 165, 166
test access point 165
test case 372
test context 147
test scenario 371
test-driven development 215
test-first approach 215
testable 136
The Open Group Architecture Framework 293
Theory of Inventive Problem Solving 90
time-to-market 14, 180
TIPS, *see* Theory of Inventive Problem Solving
TOGAF, *see* The Open Group Architecture Framework
togetherness, *see* emergence
toolbox 75
top-down 10
topology 146
Toyota Production System 218
TPS, *see* Toyota Production System
trace relationship 246, 372
traceability 49, 155, 167, 250
traceability path 81, 103
trade study 10, 69, 70, 269, 305, 341
trade-off 11, 311
training 161
TRAK 120, 292, 293
transdisciplinarity 225
transition 369
TRIZ 90
trust 98, 166, 169, 170, 175, 182, 183, 193, 306
Tuckman model 341
  forming 341
  norming 341
  performing 341
  storming 341

u

UAF, *see* OMG Unified Architecture Framework
UAF Domain Meta Model 295
UAF Profile 295
uc, *see* use case diagram
UML, *see* Unified Modeling Language
UML metamodel 351
UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification 261
Unified Modeling Language 349
Unified Profile for DoDAF and MODAF 296
unit 111
UnitAndQuantityKind library 352
unstable part 89
UPDM, *see* Unified Profile for DoDAF and MODAF
usability 10
usage 87
use case 90, 102, 105, 112, 117, 119, 171, 239, 245, 364
  continuous 107
  slice 117

system 106
use case 2.0 117
use case activity 109, 238, 240, 244
use case analysis 99
use case behavior 366
use case diagram 363, 364
use case perspective 106
use case step 109
user 161
user interface 249, 360
user perspective 105

V

V-chart, *see* V-Model
V-diagram, *see* V-Model
V-Model 78, 381–390
as process description 386
assurance-v 383, 388
basic-v 383
change 388
concurrent engineering 388
development-v 383
dissatisfaction prevention 388
dynamic-v 383
history 381
incremental development 387
iterations 387
levels 384
no comprehensive process description 383
no waterfall 386
reading instructions 389
V-Modell XT® 383
V-Modell® V-Modell® 382
value stream 383
verification planning 388
zigzag pattern, *see* zigzag pattern
validation 16, 203
value property 354
variant 269, 272, 273
definition 271
variant configuration 272, 275
definition 271
variant constraint 272, 274
definition 271
variant element 272, 274
definition 270
variant modeling 271
variation 272, 273
definition 271
variation discriminator 273
variation point 244, 272
definition 271
verification 163, 166, 171, 319
verification context 105, 147, 148
verification planning 165
verify relationship 362
video conference 337
view 48, 90, 97, 119, 167, 204, 285, 352
vs. architecture view 119
in architecture description, *see* architecture view
viewpoint 53, 120, 159, 291, 297
in architecture description, *see* architecture viewpoint
Virtual Museum Tour system 5
Virtual Tour system 7
visualization 16, 62, 66, 67, 205
visualization kind 62–64, 66, 67
Vitruvius Pollio 32
VMT, *see* Virtual Museum Tour system
VT, *see* Virtual Tour system

W

waterfall model 216, 219
Waterloo 345
what/how 76

white board 259, 338
wiki 336
work breakdown 171
work-load 97, 172, 176, 318, 321

X

XOR 271, 274

Z

Zachman Framework 285
zigzag pattern 76, 82, 163, 262, 384

WILEY SERIES IN SYSTEMS ENGINEERING AND MANAGEMENT

William Rouse, Series Editor
Andrew P. Sage, Founding Editor

ANDREW P. SAGE and JAMES D. PALMER
Software Systems Engineering

WILLIAM B. ROUSE
Design for Success: A Human-Centered Approach to Designing Successful Products and Systems

LEONARD ADELMAN
Evaluating Decision Support and Expert System Technology

ANDREW P. SAGE
Decision Support Systems Engineering

YEFIM FASSER and DONALD BRETINER
Process Improvement in the Electronics Industry, Second Edition

WILLIAM B. ROUSE
Strategies for Innovation

ANDREW P. SAGE
Systems Engineering

HORST TEMPELMEIER and HEINRICH KUHN
Flexible Manufacturing Systems: Decision Support for Design and Operation

WILLIAM B. ROUSE
Catalysts for Change: Concepts and Principles for Enabling Innovation

UPING FANG, KEITH W. HIPEL, and D. MARC KILGOUR
Interactive Decision Making: The Graph Model for Conflict

DAVID A. SCHUM
Evidential Foundations of Probabilistic Reasoning

JENS RASMUSSEN, ANNELISE MARK PEJTERSEN, and LEONARD P. GOODSTEIN
Cognitive Systems Engineering

ANDREW P. SAGE
Systems Management for Information Technology and Software Engineering

ALPHONSE CHAPANIS
Human Factors in Systems Engineering

YACOV Y. HAIMES
Risk Modeling, Assessment, and Management, Third Edition

DENNIS M. BUEDE
The Engineering Design of Systems: Models and Methods, Second Edition

ANDREW P. SAGE and JAMES E. ARMSTRONG, Jr.
Introduction to Systems Engineering

WILLIAM B. ROUSE
Essential Challenges of Strategic Management

YEFIM FASSER and DONALD BRETTNER
Management for Quality in High-Technology Enterprises

THOMAS B. SHERIDAN
Humans and Automation: System Design and Research Issues

ALEXANDER KOSSIAKOFF and WILLIAM N. SWEET
Systems Engineering Principles and Practice

HAROLD R. BOOHER
Handbook of Human Systems Integration

JEFFREY T. POLLOCK and RALPH HODGSON
Adaptive Information: Improving Business Through Semantic Interoperability, Grid Computing, and Enterprise Integration

ALAN L. PORTER and SCOTT W. CUNNINGHAM
Tech Mining: Exploiting New Technologies for Competitive Advantage

REX BROWN
Rational Choice and Judgment: Decision Analysis for the Decider

WILLIAM B. ROUSE and KENNETH R. BOFF (editors)
Organizational Simulation

HOWARD EISNER
Managing Complex Systems: Thinking Outside the Box

STEVE BELL
Lean Enterprise Systems: Using IT for Continuous Improvement

J. JERRY KAUFMAN and ROY WOODHEAD
Stimulating Innovation in Products and Services: With Function Analysis and Mapping

WILLIAM B. ROUSE
Enterprise Transformation: Understanding and Enabling Fundamental Change

JOHN E. GIBSON, WILLIAM T. SCHERER, and WILLAM F. GIBSON
How to Do Systems Analysis

WILLIAM F. CHRISTOPHER
Holistic Management: Managing What Matters for Company Success

WILLIAM B. ROUSE
People and Organizations: Explorations of Human-Centered Design

MO JAMSHIDI
System of Systems Engineering: Innovations for the Twenty-First Century

ANDREW P. SAGE and WILLIAM B. ROUSE
Handbook of Systems Engineering and Management, Second Edition

JOHN R. CLYMER
Simulation-Based Engineering of Complex Systems, Second Edition

KRAG BROTBY
Information Security Governance: A Practical Development and Implementation Approach

JULIAN TALBOT and MILES JAKEMAN
Security Risk Management Body of Knowledge

SCOTT JACKSON
Architecting Resilient Systems: Accident Avoidance and Survival and Recovery from Disruptions

JAMES A. GEORGE and JAMES A. RODGER
Smart Data: Enterprise Performance Optimization Strategy

YORAM KOREN
The Global Manufacturing Revolution: Product-Process-Business Integration and Reconfigurable Systems

AVNER ENGEL
Verification, Validation, and Testing of Engineered Systems

WILLIAM B. ROUSE (editor)
The Economics of Human Systems Integration: Valuation of Investments in People's Training and Education, Safety and Health, and Work Productivity

ALEXANDER KOSSIAKOFF, WILLIAM N. SWEET, SAMUEL J. SEYMOUR, and STEVEN M. BIEMER
Systems Engineering Principles and Practice, Second Edition

GREGORY S. PARNELL, PATRICK J. DRISCOLL, and DALE L. HENDERSON (editors)
Decision Making in Systems Engineering and Management, Second Edition

ANDREW P. SAGE and WILLIAM B. ROUSE
Economic Systems Analysis and Assessment: Intensive Systems, Organizations, and Enterprises

BOHDAN W. OPPENHEIM
Lean for Systems Engineering with Lean Enablers for Systems Engineering

LEV M. KLYATIS
Accelerated Reliability and Durability Testing Technology

BJOERN BARTELS, ULRICH ERMEL, MICHAEL PECHT, and PETER SANDBORN
Strategies to the Prediction, Mitigation, and Management of Product Obsolescence

LEVANT YILMAS and TUNCER OREN
Agent-Directed Simulation and Systems Engineering

ELSAYED A. ELSAYED
Reliability Engineering, Second Edition

BEHNAM MALAKOOTI
Operations and Production Systems with Multiple Objectives

MENG-LI SHIU, JUI-CHIN JIANG, and MAO-HSIUNG TU
Quality Strategy for Systems Engineering and Management

ANDREAS OPELT, BORIS GLOGER, WOLFGANG PFARL, and RALF MITTERMAYR
Agile Contracts: Creating and Managing Successful Projects with Scrum

KINJI MORI
Concept-Oriented Research and Development in Information Technology

KAILASH C. KAPUR and MICHAEL PECHT
Reliability Engineering

MICHAEL TORTORELLA
Reliability, Maintainability, and Supportability: Best Practices for Systems Engineers

DENNIS M. BUEDE and WILLIAM D. MILLER
The Engineering Design of Systems: Models and Methods, Third Edition

JOHN E. GIBSON, WILLIAM T. SCHERER, WILLIAM F. GIBSON, and MICHAEL C. SMITH
How to Do Systems Analysis: Primer and Casebook

GREGORY S. PARNELL
Trade-off Analytics: Creating and Exploring the System Tradespace

CHARLES S. WASSON
Systems Engineering Analysis, Design and Development

AVNER ENGEL
Practical Creativity and Innovation in Systems Engineering

ALEXANDER KOSSIAKOFF, SAMUEL J. SEYMOUR, DAVID A. FLANIGAN, and STEVEN M. BIEMER
Systems Engineering Principles and Practice, Third Edition

ELSAYED A. ELSAYED
Reliability Engineering, Third Edition

TIM WEILKIENS, JESKO G. LAMM, STEPHAN ROTH, and MARKUS WALKER
Model-Based System Architecture, Second Edition

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.